arXiv:cs/0612093v1 [cs.DC] 19 Dec 2006

N ::=	*Network*	P ::=	*Programs*
S, F	sensors and field	**idle**	idle
		$\mid P \mid P$	parallel composition
S ::=	*Sensors*	$\mid P\ ;\ P$	sequential composition
off	termination	$\mid t.v[\vec{v}]$	method invocation
$\mid N \mid N$	composition	$\mid$ **install** v	module update
$\mid [P, M]_b^{p,r}$	sensor	$\mid$ **sense** $(\vec{x})$ **in** P	field sensing
$\mid [P, M]_b^{p,r}\{S\}$	broadcast sensor	$\mid$ **if** v **then** P **else** P	conditional execution
M ::=	*Modules*	v ::=	*Values*
$\{l_i = (\vec{x}_i)\, P_i\}_{i \in I}$	method collection	x	variable
		$\mid m$	field measure
t ::=	*Targets*	$\mid p$	position
net	broadcast	$\mid b$	battery capacity
$\mid$ **this**	local	$\mid M$	module

Figure 1: The syntax of CSN.

$$E \ ::= \ N \ \mid \ [P, M]_b^{p,r}\{\mid\}E \mid E$$

This section addresses the syntax and the semantics of the Calculus for Sensor Networks. The syntax of the calculus is given by the grammar in Figure 1. The calculus encompasses a two-level structure: *networks* and *programs*.

Networks N are flat, unstructured collections of sensors and values.

A sensor $[P, M]_b^{p,r}$ represents an abstraction of a physical sensing device located at position p and running program P. Module M is the collection of methods that the sensor makes available for internal and for external usage. Typically this collection of methods may be interpreted as the library of functions of the tiny operating system installed in the sensor. Sensors may only broadcast values to its neighborhood sensors. Radius r_t defines the transmitting power of a sensor and specifies the border of communication: a circle centered at position p (the position of the sensor) with radius r_t. Likewise, radius r_s defines the sensing capability of the sensor, meaning that a sensor may only read values inside the circle centered at position p with radius r_s.

Values $\langle \vec{v} \rangle^p$ define the field of measures that may be sensed. A value consists of a tuple $\vec{v}$ denoting the strength of the measure at a given position p of the plane. Values are managed by the environment; in CSN there are no primitives for manipulating values, besides reading (sensing) values. We assume that the environment inserts these values in the network and update its contents. Networks are combined using the parallel composition operator $\mid$.

Processes are built from the inactive process **idle** and from ... **idle** denotes a terminated thread.

and sensing values from the environment *sensed* th

Programs P and Q may be combined in sequence, $P\ ;\ Q$, or in parallel, $P \mid Q$. The sequential composition $P\ ;\ Q$ designates a program that first executes P and then proceeds with the execution of Q. In contrast, $P \mid Q$ represents the simultaneous execution of P and Q. However we consider that sensors support only a very limited form of parallelism: P and Q do not interact during their execution;

Mutually recursive method definitions makes possible to represent infinite behaviours.

Values are the data exchanged between sensors, and are basic values b, method labels l, positions p, and modules M. Notice that the calculus in not high-order in the sense that communication of modules ...

As an example, consider a...

1 Programming Examples

In this section we present some examples, programmed in CSN, of typical operations performed on networks of sensors. Our goal is to show the expressiveness of the CSN calculus just presented and also to identify some other aspects of these networks that may be interesting to model. In the following examples, we denote as MSensor and MSink the modules installed in any of the anonymous sensors in the network and the modules installed in the sink, respectively. Note also that all sensors are assumed to have a builtin method, deploy, that is responsible for installing new modules. The intuition is that this method is part of the tiny operating system that allows sensors to react when first placed in the field. Finally, we assume in these small examples that the network layer supports *scoped flooding*. We shall see in the next section that this can be supported via software with the inclusion of state in sensors.

1.1 Ping

We start with a very simple ping program. Each sensor has a ping method that when invoked calls a method forward in the network with its position and battery charge as arguments. When the method forward is invoked by a sensor in the network, it just triggers another call to forward in the network. The sink has a distinct implementation of this method. Any incomming invocation logs the position and battery values given as arguments. So, the overall result of the call **net**.ping[] in the sink is that all reachable sensors in the network will, in principle, receive this call and will flood the network with their positions and battery charge values. These values eventually reach the sink and get logged.

```
MSensor(p,b) = {
  ping           = ()     net.forward[p,b]; net.ping[]
  forward        = (x,y)  net.forward[x,y]
```

$$\frac{M(l_i) = (\vec{x}_i)P_i \qquad b \geq \mathsf{c_{in}}}{[\,\mathbf{this}\,.l_i[\vec{v}]\ ;\ P_1 \mid P_2, M]_b^{p,r} \rightarrow_F [P_i[\vec{v}/\vec{x}_i]\ ;\ P_1 \mid P_2, M]_{b-\mathsf{c_{in}}}^{p,r}} \quad \text{(R-METHOD)}$$

$$\frac{l_i \notin \mathrm{dom}(M)}{[\,\mathbf{this}\,.l_i[\vec{v}]\ ;\ P_1 \mid P_2, M]_b^{p,r} \rightarrow_F [\,\mathbf{this}\,.l_i[\vec{v}]\ ;\ P_1 \mid P_2, M]_b^{p,r}} \quad \text{(R-NO-METHOD)}$$

$$\frac{\mathrm{d}(p,p') < r \qquad b \geq \mathsf{c_{out}}}{[\,\mathbf{net}\,.l_i[\vec{v}]\ ;\ P_1 \mid P_2, M]_b^{p,r}\{S\} \mid [P', M']_{b'}^{p',r'} \rightarrow_F [\,\mathbf{net}\,.l_i[\vec{v}]\ ;\ P_1 \mid P_2, M]_b^{p,r}\{S \mid [P' \mid \mathbf{this}\,.l_i[\vec{v}], M']_{b'}^{p',r'}\}} \quad \text{(R-BROADCAST)}$$

$$[\,\mathbf{net}\,.l_i[\vec{v}]\ ;\ P_1 \mid P_2, M]_b^{p,r}\{S\} \rightarrow_F [P_1 \mid P_2, M]_{b-\mathsf{c_{out}}}^{p,r} \mid S \quad \text{(R-RELEASE)}$$

$$\frac{b \geq \mathsf{c_{in}}}{[\,\mathbf{install}\ M'\ ;\ P_1 \mid P_2, M]_b^{p,r} \rightarrow_F [P_1 \mid P_2, M + M']_{b-\mathsf{c_{in}}}^{p,r}} \quad \text{(R-INSTALL)}$$

$$\frac{b \geq \mathsf{c_{in}}}{[\mathbf{sense}\,(\vec{x})\ \mathbf{in}\ P\ ;\ P_1 \mid P_2, M]_b^{p,r} \rightarrow_F [P[F(p)/\vec{x}]\ ;\ P_1 \mid P_2, M]_{b-\mathsf{c_{in}}}^{p,r}} \quad \text{(R-SENSE)}$$

$$\frac{S_1 \rightarrow_F S_2}{S \mid S_1 \rightarrow_F S \mid S_2} \qquad \frac{S_1 \equiv S_2 \qquad S_2 \rightarrow_F S_3 \qquad S_3 \equiv S_4}{S_1 \rightarrow_F S_4} \quad \text{(R-PARALLEL, R-STRUCTURAL)}$$

$$\frac{S \rightarrow_F S'}{S, F \rightarrow S', F} \quad \text{(R-NETWORK)}$$

Figure 2: Reduction semantics for processes and networks.

```
}
MSink(p,b)    = {
  forward          = (x,y) log_position_and_power[x,y]
}
[net.ping[], MSink(p,b)]_b^{p,r} |
[idle, MSensor(p_1,b_1)]_{b_1}^{p_1,r_1} | ... | [idle, MSensor(p_n,b_n)]_{b_n}^{p_n,r_n}
```

1.2 Querying

This example shows how we can program a network with a sink that periodically queries the network for the readings of the sensors. Each sensor has a sample method that samples the field using the **sense** construct and calls the method forward in the neighbourhood with its position and the value sampled as arguments. The call then queries the neighbourhood recursively with a replica

of the original call. The original call is, of course, made from the sink, which has a method start_sample that calls the method sample in the network within a cycle. Note that, if the sink had a method named sample instead of start_sample, it might get a call to sample from elsewhere in the network that could interfere with the sampling control cycle.

```
MSensor(p)    = {
  sample          = ()      sense (x) in net.forward[p, x]; net.sample[]
  forward         = (x, y)  net.forward[x,y]
}
MSink(p)      = {
  start_sample    = ()      net.sample[]; this.start_sample[]
  forward         = (x, y)  log_position_and_value[x,y]
}
[this.start_sample[], MSink(p)] b^{p,r} |
[idle, MSensor(p_1)] b_1^{p_1,r_1}  | ... | [idle, MSensor(p_n)] b_n^{p_n,r_n}
```

1.3 Polling

In this example the cycle of the sampling is done in each sensor, instead of in the sink, as in the previous example. The sink just invokes the method start_sample once. This method propagates the call through the network and invokes sample, for each sensor. This method samples the field, within a cycle, and forwards the result to the network. This implementation requires less broadcasts than the previous one as the sink only has to call start_sample on the network once. On the other hand, it increases the amount of processing per sensor.

```
MSensor(p)    = {
  start_sample    = ()      net.start_sample[]; this.sample[]
  sample          = ()      sense (x) in net.forward[p, x]; this.sample[]
  forward         = (x, y)  net.forward[x, y]
}
MSink(p)      = {
  forward         = (x, y)  log_position_and_value[x,y]
}
[net.start_example[], MSink(p)] b^{p,r} |
[idle, MSensor(p_1)] b_1^{p_1,r_1}  | ... | [idle, MSensor(p_n)] b_n^{p_n,r_n}
```

1.4 Code deployment

The above examples assume we have some means of deploying the code to the sensors. In this example we address this problem and show how it can be programmed in CSN. The code we wish to deploy and execute is the same as the one in the previous example. To achieve this goal, the sink first calls the deploy method on the network to install the new module with the methods start_sample, sample and forward as above. This call recursively deploys the code to the sensors

in the network. The sink then calls start_sample to start the sampling, again as above, and waits for the forwarded results on the method forward.

```
MSensor(p)    = {
  deploy          = (x)     install x; net.deploy[x]
}
MSink(p)      = {
  forward         = (x,y)   log_position_and_value[x,y]
}
[net.deploy[{
  start_sample    = ()      net.start_sample[]; this.sample[]
  sample          = ()      sense (x) in net.forward[p, x]; this.sample[]
  forward         = (x, y)  net.forward[x, y]
 }];
 net.start_sample[], MSink(p)] b^{p,r} |
[idle, MSensor(p1)] b1^{p1,r1}  | ... | [idle, MSensor(pn)] bn^{pn,rn}
```

A refined version of this code, one that avoids the start_sample method completely, can be programmed. Here, we deploy the code for all sensors by sending methods sample and forward to all the sensors in the network by invoking deploy. Once deployed, the code is activated with a call to sample in the sink, instead of using the start_sample method as above.

```
MSensor(p)    = {
  deploy          = (x)     install x; net.deploy[x]
}
MSink(p)      = {
  forward         = (x,y)   log_position_and_value[x,y]
}
[net.deploy[{
  sample          = ()      net.sample[];
                            install {sample = () sense (x) in net.forward[p, x];
                                                 this.sample[]};
                            this.sample[]
  forward         = (x, y)  net.forward[x, y]
 }];
 net.sample[], MSink(p)] b^{p,r} |
[idle, MSensor(p1)] b1^{p1,r1}  | ... | [idle, MSensor(pn)] bn^{pn,rn}
```

Notice that the implementation of the method sample has changed. Here, when the method is executed for the first time at each sensor, it starts by propagating the call to its neighborhood and then, it changes itself through an **install** call. The newly installed code of sample is the same as the one in the first implementation of the example. The method then continues to execute and calls the new version of sample, which starts sampling the field and forwarding values.

1.5 Sealing sensors

This example shows how we can install a sensor network with a module that contains a method, seal, that prevents any further dynamic re-programming of

the sensors, preventing anyone from tampering with the installed code. The module also contains a method, unseal that restores the original deploy method, thus allowing dynamic re-programming again. The sink just installs the module containning these methods in the network by broadcasting a method call to deploy. Each sensor that receives the call, installs the module and floods the neighborhood with a replica of the call. Another message by the sink then replaces the deploy method itself and re-implements it to **idle** . This prevents any further instalation of software in the sensors and thus effectively seals the network from external interaction other than the one allowed by the remainder of the methods in the modules of the sensors.

```
MSensor       = {
  deploy      = (x) install x; net.deploy[x]
}
MSink         = { }
[net.deploy[{
  seal        = ()  install {deploy = ()  idle}
  unseal      = ()  install {deploy = (x) install x; net.deploy[x]}
 }];
 net.seal[], MSink]^{p,r}_b |
[idle, MSensor]^{p_1,r_1}_{b_1} | ... | [idle, MSensor]^{p_n,r_n}_{b_n}
```

A Calculus for Sensor Networks

Miguel S. Silva*, Francisco Martins†, Luís Lopes*, and João Barros*

*Departamento de Ciência de Computadores & LIACC
Faculdade de Ciências da Universidade do Porto, Portugal.
†Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa, Portugal.

arXiv:cs/0612093v1 [cs.DC] 19 Dec 2006

Abstract

We consider the problem of providing a rigorous model for programming wireless sensor networks. Assuming that collisions, packet losses, and errors are dealt with at the lower layers of the protocol stack, we propose a Calculus for Sensor Networks (CSN) that captures the main abstractions for programming applications for this class of devices. Besides providing the syntax and semantics for the calculus, we show its expressiveness by providing implementations for several examples of typical operations on sensor networks. Also included is a detailed discussion of possible extensions to CSN that enable the modeling of other important features of these networks such as sensor state, sampling strategies, and network security.

keywords: Sensor Networks, Ad-Hoc Networks, Ubiquitous Computing, Process-Calculi, Programming Languages.

I. Introduction

A. The Sensor Network Challenge

Sensor networks, made of tiny, low-cost devices capable of sensing the physical world and communicating over radio links [3], are significantly different from other wireless networks: (a) the design of a sensor network is strongly driven by its particular application, (b) sensor nodes are highly constrained in terms of power consumption and computational resources (CPU, memory), and (c) large-scale sensor applications require self-configuration and distributed software updates without human intervention. Previous work on fundamental aspects of wireless sensor networks has mostly focused on communication-oriented models, in which the sensor nodes are assumed to store and process the data, coordinate their transmissions, organize the routing of messages within the network, and relay the data to a remote receiver (see *e.g.* [4,

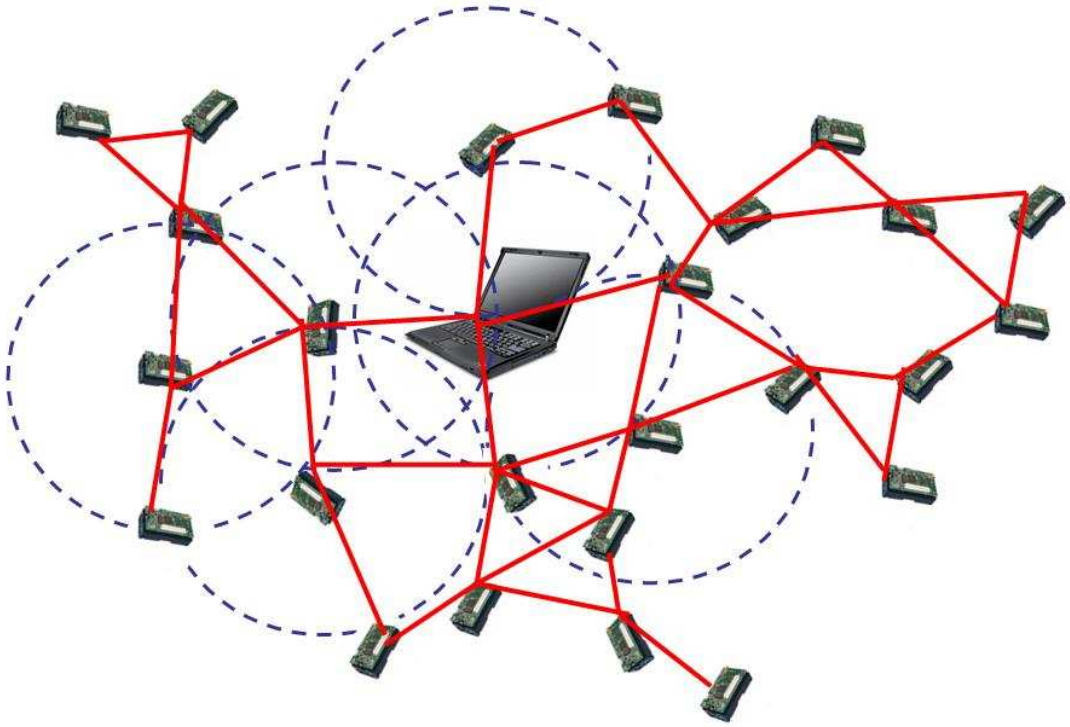

Fig. 1. A wireless sensor network is a collection of small devices that, once deployed on a target area, organize themselves in an ad-hoc network, collect measurements of a physical process and transmit the data over the wireless medium to a data fusion center for further processing.

14, 24] and references therein). Although some of these models provide useful insights (*e.g.* into the connectivity characteristics or the overall power efficiency of sensor networks) there is a strong need for formal methods that capture the inherent processing and memory constraints, and illuminate the massively parallel nature of the sensor nodes' in-network processing. If well adapted to the specific characteristics of sensor networks, a formalism of this kind, specifically a process calculus, is likely to have a strong impact on the design of operating systems, communication protocols, and programming languages for this class of distributed systems.

In terms of hardware development, the state-of-the-art is well represented by a class of multi-purpose sensor nodes called *motes*[1] [8], which were originally developed at UC Berkeley and are being deployed and tested by several research groups and start-up companies. In most of the currently available imple-

[1]Trademark of Crossbow Technology, Inc.

mentations, the sensor nodes are controlled by module-based operating systems such as TinyOS [1] and programming languages like nesC [11] or TinyScript/Maté [17]. In our view, the programming models underlying most of these tools have one or more of the following drawbacks:

1) they do not provide a rigorous model (or a calculus) of the sensor network at the programming level, which would allow for a formal verification of the correctness of programs, among other useful analysis;
2) they do not provide a global vision of a sensor network application, as a specific distributed application, making it less intuitive and error prone for programmers;
3) they require the programs to be installed on each sensor individually, something unrealistic for large sensor networks;
4) they do not allow for dynamic re-programming of the network.

Recent middleware developments such as Deluge [15] and Agilla [9] address a few of these drawbacks by providing higher level programming abstractions on top of TinyOS, including massive code deployment. Nevertheless, we are still far from a comprehensive programming solution with strong formal support and analytical capabilities.

The previous observation motivates us to design a sensor network programming model from scratch. Beyond meeting the challenges of network-wide programming and code deployment, the model should be capable of producing quantitative information on the amount of resources required by sensor network programs and protocols, and also of providing the necessary tools to prove their correctness.

B. Related Work

Given the distributed and concurrent nature of sensor network operations, we build our sensor network calculus on thirty years of experience gathered by concurrency theorists and programming language designers in pursuit of an adequate formalism and theory for concurrent systems. The first steps towards this goal were given by Milner [20] with the development of CCS (Calculus of Communicating Systems). CCS describes computations in which concurrent processes may interact through simple synchronization, without otherwise exchanging information. Allowing processes to exchange resources (*e.g.*, links, memory references, sockets, code), besides synchronizing, considerably increases the expressive power of the formal systems. Such systems, known as process-calculi, are able to model the mobility patterns of the resources and thus constitute valuable tools to reason about concurrent, distributed systems.

The first such system, built on Milner's work, was the π-calculus [21]. Later developments of this initial proposal allowed for further simplification an provided an asynchronous form of the calculus [5, 13]. Since then, several calculi have been proposed to model concurrent distributed systems and for many there are prototype implementations of programming languages and run-time systems (*e.g.* Join [10], TyCO [27], X-Klaim [16], and Nomadic Pict [28]).

Previous work by Prasad [23] established the first process calculus approach to modeling broadcast based systems. Later work by Ostrovský, Prasad, and Taha [22] established the basis for a higher-order calculus for broadcasting systems. The focus of this line of work lies in the protocol layer of the networks, trying to establish an operational semantics and associated theory that allows assertions to be made about the networks. More recently, Mezzetti and Sangiorgi [19] discuss the use of process calculi to model wireless systems, again focusing on the details of the lower layers of the protocol stack (*e.g.* collision avoidance) and establishing an operational semantics for the networks.

C. Our Contributions

Our main contribution is a sensor network programming model based on a process calculus, which we name Calculus of Sensor Networks (CSN). Our calculus offers the following features that are specifically tailored for sensor networks:

- *Top-Level Approach*: CSN focuses on programming and managing sensor networks and so it assumes that collisions, losses, and errors have been dealt with at the lower layers of the protocol stack and system architecture (this distinguishes CSN from the generic wireless network calculus presented in [19]);
- *Scalability*: CSN offers the means to provide the sensor nodes with self-update and self-configuration abilities, thus meeting the challenges of programming and managing a large-scale sensor network;
- *Broadcast Communication*: instead of the peer-to-peer (unicast) communication of typical process calculi, CSN captures the properties of broadcast communication as favored by sensor networks (with strong impact on their energy consumption);
- *Ad-hoc Topology*: network topology is not required to be programmed in the processes, which would be unrealistic in the case of sensor networks;
- *Communication Constraints*: due to the power limitations of their wireless interface, the sensor nodes can only communicate with their direct neighbors in the network and thus the notion of neighborhood

of a sensor node, *i.e.* the set of sensor nodes within its communication range, is introduced directly in the calculus;

- *Memory and Processing Constraints*: the typical limitations of sensor networks in terms of memory and processing capabilities are captured by explicitly modeling the internal processing (or the *intelligence*) of individual sensors;
- *Local Sensing*: naturally, the sensors are only able to pick up local measurements of their environment and thus have geographically limited sensitivity.

To provide these features, we devise CSN as a two-layer calculus, offering abstractions for data acquisition, communication, and processing. The top layer is formed by a network of sensor nodes immersed in a scalar or vector field (representing the physical process captured by the sensor nodes). The sensor nodes are assumed to be running in parallel. Each sensor node is composed of a collection of labeled methods, which we call a module, and that represents the code that can be executed in the device. A process is executed in the sensor node as a result of a remote procedure call on a module by some other sensor or, seen from the point of view of the callee, as a result of the reception of a message. Sensor nodes are multithreaded and may share state, for example, in a tuple-space. Finally, by adding the notions of position and range, we are able to capture the nature of broadcast communication and the geographical limits of the sensor network applications.

The remainder of this paper is structured as follows. The next section describes the syntax and semantics of the CSN calculus. Section III presents several examples of functionalities that can be implemented using CSN and that are commonly required in sensor networks. In Section IV we discuss some design options we made and how we can extend CSN to model other aspects of sensor networks. Finally, Section V presents some conclusions and directions for future work.

II. The Calculus

This section addresses the syntax and the semantics of the Calculus for Sensor Networks. For simplicity, in the remainder of the paper we will refer to a sensor node or a sensor device in a network as a *sensor*. The syntax is provided by the grammar in Figure 2, and the operational semantics is given by the reduction relation depicted in Figures 3 and 4.

$N ::=$	*Network*	$P ::=$	*Programs*
S, F	sensors and field	**idle**	idle
		$\mid P \mid P$	parallel composition
$S ::=$	*Sensors*	$\mid P \,;\, P$	sequential composition
off	termination	$\mid t.v[\vec{v}]$	method invocation
$\mid N \mid N$	composition	$\mid$ **install** v	module update
$\mid [P, M]_b^{p,r}$	sensor	$\mid$ **sense** $(\vec{x})$ **in** P	field sensing
$\mid [P, M]_b^{p,r}\{S\}$	broadcast sensor	$\mid$ **if** v **then** P **else** P	conditional execution
$M ::=$	*Modules*	$v ::=$	*Values*
$\{l_i = (\vec{x}_i)\, P_i\}_{i \in I}$	method collection	x	variable
		$\mid m$	field measure
$t ::=$	*Targets*	$\mid p$	position
net	broadcast	$\mid b$	battery capacity
$\mid$ **this**	local	$\mid M$	module

Fig. 2. The syntax of CSN.

A. Syntax

Let $\vec{\alpha}$ denote a possible empty sequence $\alpha_1 \ldots \alpha_n$ of elements of the syntactic category α. Assume a countable set of *labels*, ranged over by letter l, used to name methods within modules, and a countable set of *variables*, disjoint from the set of labels and ranged over by letter x. Variables stand for communicated values (*e.g.* battery capacity, position, field measures, modules) in a given program context.

The syntax for CNS is found in Figure 2. We explain the syntactic constructs along with their informal, intuitive semantics. Refer to the next section for a precise semantics of the calculus.

Networks N denote the composition of sensor networks S with a (scalar or vector) field F. A field is

a set of pairs (position, measure) describing the distribution of some physical quantity (*e.g.* temperature, pressure, humidity) in space. The position is given in some coordinate system. Sensors can measure the intensity of the field in their respective positions.

Sensor networks S are flat, unstructured collections of sensors combined using the parallel composition operator.

A sensor $[P, M]_b^{p,r}$ represents an abstraction of a physical sensing device and is parametric in its position p, describing the location of the sensor in some coordinate system; its transmission range specified by the radius r of a circle centered at position p; and its battery capacity b. The position of the sensors may vary with time if the sensor is mobile in some way. The transmission range, on the other hand, usually remains constant over time. A sensor with the battery exhausted is designated by **off**.

Inside a sensor there exists a running program P and a module M. A module is a collection of methods defined as $l = (\vec{x})P$ that the sensor makes available for internal and for external usage. A method is identified by label l and defined by an abstraction $(\vec{x})P$: a program P with parameters $\vec{x}$. Method names are pairwise distinct within a module. Mutually recursive method definitions make it possible to represent infinite behavior. Intuitively, the collection of methods of a sensor may be interpreted as the function calls of some tiny operating system installed in the sensor.

Communication in the sensor network only happens via broadcasting values from one sensor to its neighborhood: the sensors inside a circle centered at position p (the position of the sensor) with radius r. A broadcast sensor $[P, M]_b^{p,r}\{S\}$ stands for a sensor during the broadcast phase, having already communicated with sensors S. While broadcasting, it is fundamental to keep track of the sensors engaged in communication so far, thus preventing the delivery of the same message to the same sensor during one broadcasting operation. Target sensors are collected in the *bag* of the sensor emitting the message. Upon finishing the broadcast the bag is emptied out, and the (target) sensors are released into the network. This construct is a run-time construct and is available to the programmer.

Programs are ranged over by P. The **idle** program denotes a terminated thread. Method invocation, $t.v[\vec{v}]$, selects a method v (with arguments $\vec{v}$) either in the local module or broadcasts the request to the neighborhood sensors, depending whether t is the keyword **this** or the keyword **net**, respectively. Program **sense** $(\vec{x})$ **in** P reads a measure from the surrounding field and binds it to $\vec{x}$ within P. Installing or replacing methods in the sensor's module is performed using the construct **install** v. The calculus also offers a standard form of branching through the **if** v **then** P **else** P construct.

Programs P and Q may be combined in sequence, $P \;;\; Q$, or in parallel, $P \,|\, Q$. The sequential

composition $P\ ;\ Q$ designates a program that first executes P and then proceeds with the execution of Q. In contrast, $P \mid Q$ represents the simultaneous execution of P and Q.

Values are the data exchanged between sensors and comprise field measures m, positions p, battery capacities b, and modules M. Notice that this is not a higher-order calculus: communicating a module means the ability to transfer its *code* to, to retransmit it from, or to install it in a remote sensor.

B. Examples

Our first example illustrates a network of sensors that sample the field and broadcast the measured values to a special node known as the *sink*. The sink node may be no different from the other sensors in the network, except that it usually possesses a distinct software module that allows it to collect and process the values broadcasted in the network. The behavior we want to program is the following. The sink issues a request to the network to sample the field; upon reception of the request each sensor samples the field at its position and broadcasts the measured value back to the sink; the sink receives and processes the values. An extended version of this example may be found in Section III-B.

The code for the modules of the sensors, MSensor(p, r), and for the sink, MSink(p, r), is given below. Both modules are parametric in the position and in the broadcasting range of each sensor.

As for the module equipping the sensors, it has a method sample that, when invoked, propagates the call to its neighborhood (**net**.sample[];), samples the field (**sense** x **in** ...) and forwards the value to the network (... **net**.forward[p,x]). Notice that each sensor propagates the original request from the sink. This is required since in general most of the sensors in the network will be out of broadcasting range from the sink. Therefore each sensor echos the request, hopefully covering all the network. Message forwarding will be a recurrent pattern found in our examples. Another method of the sensors' module is forward that simply forwards the values from other sensors through the network.

The module for the sink contains a different implementation of the forward method, since the sink will gather the values sent by the sensors and will log them. Here we leave unspecified the processing done by the log_position_and_value program.

The network starts-up with all sensors **idle**, except for the sink that requests a sampling (**net**.sample[]).

```
MSensor(p,r) = { sample  = () net.sample[]; sense x in net.forward[p,x]
                 forward = (x,y) net.forward[x,y] }
MSink(p,r)   = { forward = (x,y) log_position_and_value[x,y] }
```

```
[net.sample[], MSink(p,r)] b^{p,r} |
[idle, MSensor(p1,r1)] b1^{p1,r1} | ... | [idle, MSensor(pn,rn)] bn^{pn,rn}
```

The next example illustrates the broadcast, the deployment, and the installation of code. The example runs as follows. The sink node deploys some module in the network (**net**.deploy[M]) and then seals the sensors (**net**.seal[]), henceforth preventing any dynamic re-programming of the network. An extended version of the current example may be found in Section III-E.

The code for the modules of the sensors and of the sink is given below. The module M is the one we wish to deploy to the network. It carries the method seal that forwards the call to the network and installs a new version of deploy that does nothing when executed.

```
MSensor(p,r) = { deploy = (x) net.deploy[x]; install x }
MSink(p,r)   = {}
M            = { seal = () net.seal[]; install { deploy = () idle } }

[net.deploy[M]; net.seal[], MSink(p,r)] b^{p,r} |
[idle, MSensor(p1,r1)] b1^{p1,r1} | ... | [idle, MSensor(pn,rn)] bn^{pn,rn}
```

C. Semantics

The calculus has two name bindings: field sensing and method definitions. The displayed occurrence of name x_i is a *binding* with *scope* P both in **sense** $(x_1, \ldots, x_i, \ldots, x_n)$ **in** P and in $l = (x_1, \ldots, x_i, \ldots, x_n)\, P$. An occurrence of a name is *free* if it is not in the scope of a binding. Otherwise, the occurrence of the name is *bound*. The set of free names of a sensor S is referred as $\mathrm{fn}(S)$.

Following Milner [20] we present the reduction relation with the help of a structural congruence relation. The structural congruence relation $\equiv$, depicted in Figure 3, allows for the manipulation of term structure, adjusting sub-terms to reduce. The relation is defined as the smallest congruence relation on sensors (and programs) closed under the rules given in Figure 3.

The parallel composition operators for programs and for sensors are taken to be commutative and associative with **idle** and **off** as their neutral elements, respectively (*vide* Rules S-MONOID-PROGRAM and S-MONOID-SENSOR). Rule S-IDLE-SEQ asserts that **idle** is also neutral with respect to sequential composition of programs. Rule S-PROGRAM-STRU incorporates structural congruence for programs into sensors. When a sensor is broadcasting a message it uses a bag to collect the sensors as they become

$$P_1 \mid P_2 \equiv P_2 \mid P_1, \qquad P \mid \textbf{idle} \equiv P, \qquad P_1 \mid (P_2 \mid P_3) \equiv (P_1 \mid P_2) \mid P_3 \qquad \text{(S-MONOID-PROGRAM)}$$

$$S_1 \mid S_2 \equiv S_2 \mid S_1, \qquad S \mid \textbf{off} \equiv S, \qquad S_1 \mid (S_2 \mid S_3) \equiv (S_1 \mid S_2) \mid S_3 \qquad \text{(S-MONOID-SENSOR)}$$

$$\textbf{idle} \ ; \ P \equiv P \qquad \frac{P_1 \equiv P_2}{[P_1, M]_b^{p,r} \equiv [P_2, M]_b^{p,r}} \qquad \text{(S-IDLE-SEQ, S-PROGRAM-STRU)}$$

$$[P, M]_b^{p,r} \equiv [P, M]_b^{p,r}\{\textbf{off}\} \qquad \frac{b < \max(\mathsf{c_{in}}, \mathsf{c_{out}})}{[P, M]_b^{p,r} \equiv \textbf{off}} \qquad \text{(S-BROADCAST, S-BAT-EXHAUSTED)}$$

Fig. 3. Structural congruence for processes and sensors.

engaged in communication. Rule S-BROADCAST allows for a sensor to start the broadcasting operation. A terminated sensor is a sensor with insufficient battery capacity for performing an internal or an external reduction step (*vide* Rule S-BAT-EXHAUSTED).

The reduction relation on networks, notation $S, F \rightarrow S', F$, describes how sensors S can evolve (reduce) to sensors S', sensing the field F. The reduction is defined on top of a reduction relation for sensors, notation $S \rightarrow_F S'$, inductively defined by the rules in Figure 4. The reduction for sensors is parametric on field F and on two constants $\mathsf{c_{in}}$ and $\mathsf{c_{out}}$ that represent the amount of energy consumed when performing internal computation steps ($\mathsf{c_{in}}$) and when broadcasting messages ($\mathsf{c_{out}}$).

Computation inside sensors proceeds by invoking a method (either local—Rules R-METHOD and R-NO-METHOD—or remote—Rules R-BROADCAST and R-RELEASE), by sensing values (Rule R-SENSE), and by updating the method collection of the sensor (Rule R-INSTALL).

The invocation of a local method l_i with arguments $\vec{v}$ evolves differently depending on whether or not the definition for l_i is part of the method collection of the sensor. Rule R-METHOD describes the invocation of a method from module M, defined as $M(l_i) = (\vec{x}_i)P_i$. The result is the program P_i where the values $\vec{v}$ are bound to the variables in $\vec{x}$. When the definition for l_i is not present in M, we have decided to actively wait for the definition (see Rule R-NO-METHOD). Usually invoking an undefined method causes a program to get *stuck*. Typed programming languages use a type system to ensure that there are no invocations to undefined methods, ruling out all other programs at compile time. At run-time, another possible choice would be to simply discard invocations to undefined methods. Our choice provides more resilient applications when coupled with the procedure for deploying code in a sensor

$$\frac{M(l_i) = (\vec{x}_i)P_i \qquad b \geq \mathsf{c_{in}}}{[\mathbf{this}\,.l_i[\vec{v}]\ ;\ P_1 \mid P_2, M]_b^{p,r} \rightarrow_F [P_i[\vec{v}/\vec{x}_i]\ ;\ P_1 \mid P_2, M]_{b-\mathsf{c_{in}}}^{p,r}} \quad \text{(R-METHOD)}$$

$$\frac{l_i \notin \mathrm{dom}(M)}{[\mathbf{this}\,.l_i[\vec{v}]\ ;\ P_1 \mid P_2, M]_b^{p,r} \rightarrow_F [\mathbf{this}\,.l_i[\vec{v}]\ ;\ P_1 \mid P_2, M]_b^{p,r}} \quad \text{(R-NO-METHOD)}$$

$$\frac{\mathrm{d}(p, p') < r \qquad b \geq \mathsf{c_{out}}}{\begin{array}{l}[\mathbf{net}\,.l_i[\vec{v}]\ ;\ P_1 \mid P_2, M]_b^{p,r}\{S\} \mid [P', M']_{b'}^{p',r'} \rightarrow_F \\ \qquad [\mathbf{net}\,.l_i[\vec{v}]\ ;\ P_1 \mid P_2, M]_b^{p,r}\{S \mid [P' \mid \mathbf{this}\,.l_i[\vec{v}], M']_{b'}^{p',r'}\}\end{array}} \quad \text{(R-BROADCAST)}$$

$$[\mathbf{net}\,.l_i[\vec{v}]\ ;\ P_1 \mid P_2, M]_b^{p,r}\{S\} \rightarrow_F [P_1 \mid P_2, M]_{b-\mathsf{c_{out}}}^{p,r} \mid S \quad \text{(R-RELEASE)}$$

$$\frac{b \geq \mathsf{c_{in}}}{[\mathbf{install}\ M'\ ;\ P_1 \mid P_2, M]_b^{p,r} \rightarrow_F [P_1 \mid P_2, M + M']_{b-\mathsf{c_{in}}}^{p,r}} \quad \text{(R-INSTALL)}$$

$$\frac{b \geq \mathsf{c_{in}}}{[\mathbf{sense}\ (\vec{x})\ \mathbf{in}\ P\ ;\ P_1 \mid P_2, M]_b^{p,r} \rightarrow_F [P[F(p)/\vec{x}]\ ;\ P_1 \mid P_2, M]_{b-\mathsf{c_{in}}}^{p,r}} \quad \text{(R-SENSE)}$$

$$\frac{S_1 \rightarrow_F S_2}{S \mid S_1 \rightarrow_F S \mid S_2} \qquad \frac{S_1 \equiv S_2 \qquad S_2 \rightarrow_F S_3 \qquad S_3 \equiv S_4}{S_1 \rightarrow_F S_4} \quad \text{(R-PARALLEL, R-STRUCTURAL)}$$

$$\frac{S \rightarrow_F S'}{S, F \rightarrow S', F} \quad \text{(R-NETWORK)}$$

Fig. 4. Reduction semantics for processes and networks.

network. We envision that if we invoke a method in the network after some code has been deployed (see Example III-D), there may be some sensors where the method invocation arrives before the deployed code. With the semantics we propose, the call actively waits for the code to be installed.

Sensors communicate with the network by broadcasting messages. A message consists of a remote method invocation on unspecified sensors in the neighborhood of the emitting sensor. In other words, the messages are not targeted to a particular sensor (there is no peer-to-peer communication). The neighborhood of a sensor is defined by its communication radius, but there is no guarantee that a message broadcasted by a given sensor arrives at all surrounding sensors. There might be, for instance, landscape obstacles that prevent two sensors, otherwise within range, from communicating with each other. Also,

during a broadcast operation the message must only reach each neighborhood sensor once. Notice that we are not saying that the same message can not reach the same sensor multiple times. In fact it might, but as the result of the echoing of the message in subsequent broadcast operations. We model the broadcasting of messages in two stages. Rule R-BROADCAST invokes method l_i in the remote sensor, provided that the distance between the emitting and the receiving sensors is less that the transmission radius ($\mathrm{d}(p, p') < r$). The sensor receiving the message is put in the bag of the emitting sensor, thus preventing multiple deliveries of the same message while broadcasting. Observe that the rule does not enforce the interaction with all sensors in the neighborhood. Rule R-RELEASE finishes the broadcast by consuming the operation ($\mathbf{net}.l_i[\vec{v}]$), and by emptying out the contents of the emitting sensor's bag. A broadcast operation starts with the application of Rule S-BROADCAST, proceeds with multiple (eventually none) applications of Rule R-BROADCAST (one for each target sensor), and terminates with the application of Rule R-RELEASE.

Installing module M' in a sensor with a module M, Rule R-INSTALL, amounts to add to M the methods in M' (absent in M), and to replace (in M) the methods common to both M and M'. Rigorously, the operation of installing module M' on top of M, denoted $M + M'$, may be defined as $M + M' = (M \setminus M') \cup M'$. The $+$ operator is reminiscent of Abadi and Cardelli's operator for updating methods in their imperative object calculus [2].

A sensor senses the field in which it is immersed, Rule R-SENSE, by sampling the value of the field F in its position p and, continues the computation replacing this value for the bound variables $\vec{x}$ in program P.

Rule R-PARALLEL allows reduction to happen in networks of sensors and Rule R-STRUCTURAL brings structural congruence into the reduction relation.

D. The Operational Semantics Illustrated

To illustrate the operational semantics of CNS, we present the reduction steps for the examples discussed at the end of Section II-B. During reduction we suppress the side annotations when writing the sensors. Due to space constraints we consider a rather simple network with just the sink and another sensor.

$[\,\mathbf{net}\,.\,\mathsf{sample}\,[\,]\,,\mathsf{MSink}\,(\,p\,,r\,)\,]\;\;|\;\;[\,\mathbf{idle}\,,\mathsf{MSensor}\,(p_1,r_1)\,]$

We assume that the sensor is within range from the sink and vice-versa. This network may reduce as follows:

[**net**.sample[],MSink(p,r)] | [**idle**,MSensor(p_1,r_1)] $\equiv$ (S-BROADCAST)

[**net**.sample[],MSink(p,r)]{**off**} | [**idle**,MSensor(p_1,r_1)] $\rightarrow\equiv$

($\mathrm{d}(p, p_1) < r$, R-BROADCAST, S-MONOID-SENSOR)

[**net**.sample[],MSink(p,r)] {[**this**.sample[] || **idle**,MSensor(p_1,r_1)]} $\rightarrow\equiv$

(R-RELEASE, S-MONOID-PROGRAM)

[**idle**,MSink(p,r)] | [**this**.sample[],MSensor(p_1,r_1)] $\rightarrow$ (R-METHOD)

[**idle**,MSink(p,r)] | [**net**.sample[]; **sense** x **in net**.forward[p_1,x],MSensor(p_1,r_1)] $\equiv$ (S-BROADCAST)

[**idle**,MSink(p,r)] | [**net**.sample[]; **sense** x **in net**.forward[p_1,x],MSensor(p_1,r_1)]{**off**} $\rightarrow$ (R-RELEASE)

[**idle**,MSink(p,r)] | [**sense** x **in net**.forward[p_1,x],MSensor(p_1,r_1)] $\rightarrow$ (R-SENSE)

[**idle**,MSink(p,r)] | [**net**.forward[p_1,$F(p_1)$], MSensor(p_1,r_1)] $\equiv$ (S-MONOID-SENSOR)

[**net**.forward[p_1,$F(p_1)$], MSensor(p_1,r_1)] | [**idle**,MSink(p,r)] $\equiv$ (S-BROADCAST)

[**net**.forward[p_1,$F(p_1)$], MSensor(p_1,r_1)]{**off**} | [**idle**,MSink(p,r)] $\rightarrow\equiv$

($\mathrm{d}(p_1, p) < r_1$, R-BROADCAST, S-MONOID-SENSOR)

[**net**.forward[p_1,$F(p_1)$], MSensor(p_1,r_1)]{[**this**.forward[p_1,$F(p_1)$] || **idle**, MSink(p,r)]} $\rightarrow\equiv$

(R-RELEASE, S-MONOID-PROGRAM)

[**idle**, MSensor(p_1,r_1)] | [**this**.forward[p_1,$F(p_1)$], MSink(p,r)] $\rightarrow$ (R-METHOD)

[**idle**, MSensor(p_1,r_1)] | [log_position_and_value[p_1,$F(p_1)$], MSink(p,r)] $\equiv$ (S-MONOID-SENSOR)

[log_position_and_value[p_1,$F(p_1)$] | [**idle**, MSensor(p_1,r_1)], MSink(p,r)]

So, after these reduction steps the sink gets the field values from the sensor at position p_1 and logs them. The sensor at p_1 is idle waiting for further interaction.

Following we present the reduction step for our second (and last) example of Section II-B where we illustrate the broadcast, the deployment, and the installation of code. Again, due to space restrictions, we use a very simple network with just the sink and another sensor, both within reach of each other.

[**net**.deploy[M]; **net**.seal[],MSink(p,r)] | [**idle**,MSensor(p_1,r_1)]

This network may reduce as follows:

[**net**.deploy[M]; **net**.seal [], MSink(p,r)] | [**idle**, MSensor(p_1,r_1)] $\equiv$ (S-BROADCAST)

[**net**.deploy[M]; **net**.seal [], MSink(p,r)]{**off**} | [**idle**, MSensor(p_1,r_1)] $\rightarrow\equiv$
($\mathrm{d}(p, p_1) < r$, R-BROADCAST, S-MONOID-SENSOR)

[**net**.deploy[M]; **net**.seal [], MSink(p,r)] | {[**this**.deploy[M] || **idle**, MSensor(p_1,r_1)]} $\rightarrow\equiv$
(R-RELEASE, S-MONOID-PROGRAM)

[**net**.seal [], MSink(p,r)] | [**this**.deploy[M], MSensor(p_1,r_1)] $\rightarrow$ (R-METHOD)

[**net**.seal [], MSink(p,r)] | [**net**.deploy[M]; **install** M, MSensor(p_1,r_1)] $\equiv$ (S-BROADCAST)

[**net**.seal [], MSink(p,r)] | [**net**.deploy[M]; **install** M, MSensor(p_1,r_1)]{**off**} $\rightarrow\equiv$
(R-RELEASE, S-MONOID-SENSOR)

[**net**.seal [], MSink(p,r)] | [**install** M, MSensor(p_1,r_1)] $\rightarrow$ (R-INSTALL)

[**net**.seal [], MSink(p,r)] | [**idle**, MSensor(p_1,r_1)+M] $\equiv$ (S-BROADCAST)

[**net**.seal [], MSink(p,r)]{**off**} | [**idle**, MSensor(p_1,r_1)+M] $\rightarrow\equiv$
($\mathrm{d}(p, p_1) < r$, R-BROADCAST, S-MONOID-SENSOR)

[**net**.seal [], MSink(p,r)] {[**this**.seal [] || **idle**, MSensor(p_1,r_1)+M]} $\rightarrow\equiv$
(R-RELEASE, S-MONOID-PROGRAM)

[**idle**, MSink(p,r)] | [**this**.seal [], MSensor(p_1,r_1)+M] $\rightarrow$ (R-METHOD)

[**idle**, MSink(p,r)] | [**net**.seal []; **install** {deploy = () **idle**}, MSensor(p_1,r_1)+M] $\equiv$ (S-BROADCAST)

[**idle**, MSink(p,r)] | [**net**.seal []; **install** {deploy = () **idle**}, MSensor(p_1,r_1)+M]{**off**} $\rightarrow\equiv$
(R-RELEASE, S-MONOID-SENSOR)

[**idle**, MSink(p,r)] | [**install** {deploy = () **idle**}, MSensor(p_1,r_1)+M] $\rightarrow$ (R-INSTALL)

[**idle**, MSink(p,r)] | [**idle**, MSensor(p_1,r_1)+M+{deploy = () **idle**}]

After these reductions, the sink is idle after deploying the code to the sensor at p. The sensor at p is also idle, waiting for interaction, but with the code for the module M installed and with the deploy method disabled.

III. Programming Examples

In this section we present some examples, programmed in CSN, of typical operations performed on networks of sensors. Our goal is to show the expressiveness of the CSN calculus just presented and also to identify some other aspects of these networks that may be interesting to model. In the following examples, we denote as MSensor and MSink the modules installed in any of the anonymous sensors in the network and the modules installed in the sink, respectively. Note also that all sensors are assumed to have a builtin method, deploy, that is responsible for installing new modules. The intuition is that this method is part of the tiny operating system that allows sensors to react when first placed in the field. Finally, we assume in these small examples that the network layer supports *scoped flooding*. We shall see in the next section that this can be supported via software with the inclusion of state in sensors.

A. Ping

We start with a very simple ping program. Each sensor has a ping method that when invoked calls a method forward in the network with its position and battery charge as arguments. When the method forward is invoked by a sensor in the network, it just triggers another call to forward in the network. The sink has a distinct implementation of this method. Any incomming invocation logs the position and battery values given as arguments. So, the overall result of the call **net**.ping[] in the sink is that all reachable sensors in the network will, in principle, receive this call and will flood the network with their positions and battery charge values. These values eventually reach the sink and get logged.

```
MSensor(p,b) = {
  ping           = ()     net.forward[p,b]; net.ping[]
  forward        = (x,y)  net.forward[x,y]
}
MSink(p,b)   = {
  forward        = (x,y)  log_position_and_power[x,y]
}
[net.ping[], MSink(p,b)] p,r b |
[idle, MSensor(p1,b1)] p1,r1 b1 | ... | [idle, MSensor(pn,bn)] pn,rn bn
```

B. Querying

This example shows how we can program a network with a sink that periodically queries the network for the readings of the sensors. Each sensor has a sample method that samples the field using the **sense** construct and calls the method forward in the neighbourhood with its position and the value sampled as arguments. The call then queries the neighbourhood recursively with a replica of the original call. The original call is, of course, made from the sink, which has a method start_sample that calls the method sample in the network within a cycle. Note that, if the sink had a method named sample instead of start_sample, it might get a call to sample from elsewhere in the network that could interfere with the sampling control cycle.

```
MSensor(p)    = {
  sample          = ()      sense (x) in net.forward[p, x]; net.sample[]
  forward         = (x, y)  net.forward[x,y]
}
MSink(p)      = {
  start_sample    = ()      net.sample[]; this.start_sample[]
  forward         = (x, y)  log_position_and_value[x,y]
}
[this.start_sample[], MSink(p)] b^{p,r} |
[idle, MSensor(p_1)] b_1^{p_1,r_1}  |  ...  |  [idle, MSensor(p_n)] b_n^{p_n,r_n}
```

C. Polling

In this example the cycle of the sampling is done in each sensor, instead of in the sink, as in the previous example. The sink just invokes the method start_sample once. This method propagates the call through the network and invokes sample, for each sensor. This method samples the field, within a cycle, and forwards the result to the network. This implementation requires less broadcasts than the previous one as the sink only has to call start_sample on the network once. On the other hand, it increases the amount of processing per sensor.

```
MSensor(p)    = {
  start_sample    = ()      net.start_sample[]; this.sample[]
  sample          = ()      sense (x) in net.forward[p, x]; this.sample[]
  forward         = (x, y)  net.forward[x, y]
```

```
}
MSink(p)      = {
  forward         = (x, y) log_position_and_value[x,y]
}
[net.start_example[], MSink(p)] b^{p,r} |
[idle, MSensor(p_1)] b_1^{p_1,r_1} | ... | [idle, MSensor(p_n)] b_n^{p_n,r_n}
```

D. Code deployment

The above examples assume we have some means of deploying the code to the sensors. In this example we address this problem and show how it can be programmed in CSN. The code we wish to deploy and execute is the same as the one in the previous example. To achieve this goal, the sink first calls the deploy method on the network to install the new module with the methods start_sample, sample and forward as above. This call recursively deploys the code to the sensors in the network. The sink then calls start_sample to start the sampling, again as above, and waits for the forwarded results on the method forward.

```
MSensor(p)    = {
  deploy          = (x)     install x; net.deploy[x]
}
MSink(p)      = {
  forward         = (x,y)   log_position_and_value[x,y]
}
[net.deploy[{
  start_sample    = ()      net.start_sample[]; this.sample[]
  sample          = ()      sense (x) in net.forward[p, x]; this.sample[]
  forward         = (x, y)  net.forward[x, y]
 }];
 net.start_sample[], MSink(p)] b^{p,r} |
[idle, MSensor(p_1)] b_1^{p_1,r_1} | ... | [idle, MSensor(p_n)] b_n^{p_n,r_n}
```

A refined version of this code, one that avoids the start_sample method completely, can be programmed. Here, we deploy the code for all sensors by sending methods sample and forward to all the sensors in the network by invoking deploy. Once deployed, the code is activated with a call to sample in the sink, instead of using the start_sample method as above.

```
MSensor(p)    = {
  deploy          = (x)    install x; net.deploy[x]
}
MSink(p)      = {
  forward         = (x,y) log_position_and_value[x,y]
}
[net.deploy[{
  sample          = ()     net.sample[];
                           install {sample = () sense (x) in net.forward[p, x];
                                                this.sample[]};
                           this.sample[]
  forward         = (x, y) net.forward[x, y]
 }];
 net.sample[], MSink(p)] b^{p,r} |
[idle, MSensor(p_1)] b_1^{p_1,r_1} | ... | [idle, MSensor(p_n)] b_n^{p_n,r_n}
```

Notice that the implementation of the method sample has changed. Here, when the method is executed for the first time at each sensor, it starts by propagating the call to its neighborhood and then, it changes itself through an **install** call. The newly installed code of sample is the same as the one in the first implementation of the example. The method then continues to execute and calls the new version of sample, which starts sampling the field and forwarding values.

E. Sealing sensors

This example shows how we can install a sensor network with a module that contains a method, seal, that prevents any further dynamic re-programming of the sensors, preventing anyone from tampering with the installed code. The module also contains a method, unseal that restores the original deploy method, thus allowing dynamic re-programming again. The sink just installs the module containning these methods in the network by broadcasting a method call to deploy. Each sensor that receives the call, installs the module and floods the neighborhood with a replica of the call. Another message by the sink then replaces the deploy method itself and re-implements it to **idle**. This prevents any further instalation of software in the sensors and thus effectively seals the network from external interaction other than the one allowed by the remainder of the methods in the modules of the sensors.

```
MSensor      = {
  deploy     = (x) install x; net.deploy[x]
}
MSink        = { }
[net.deploy[{
  seal       = ()  install {deploy = ()  idle}
  unseal     = ()  install {deploy = (x) install x; net.deploy[x]}
 }];
 net.seal[], MSink]_b^{p,r} |
[idle, MSensor]_{b_1}^{p_1,r_1} | ... | [idle, MSensor]_{b_n}^{p_n,r_n}
```

IV. Discussion

In the previous sections, we focused our attention on the programming issues of a sensor network and presented a core calculus that is expressive enough to model fundamental operations such as local broadcast of messages, local sensing of the environment, and software module updates. CSN allows the global modeling of sensor networks in the sense that it allows us to design and implement sensor network applications as large-scale distributed applications, rather than giving the programmer a sensor-by-sensor view of the programming task. It also provides the tools to manage running sensor networks, namely through the use of the software deployment capabilities.

There are other important features of sensor networks that we consciously left out of CSN. In the sequel we discuss some of these features and sketch some ideas of how we would include support for them.

a) State: From a programming point of view, adding state to sensors is essential. Sensors have some limited computational capabilities and may perform some data processing before sending it to the sink. This processing assumes that the sensor is capable of buffering data and thus maintain some state. In a way, CSN sensors have state. Indeed, the atributes p, b, and r may be viewed as sensor state. Since these are characteristic of each sensor and are usually controlled at the hardware level, we chose to represent this state as parameters of the sensors. The programmer may read these values at any time through builtin method calls but any change to this data is performed transparently for the programmer by the hardware or operating system. As we mentioned before, it is clear that the value of b changes with time. The

position p may also change with time if we envision our sensors endowed with some form of mobility (*e.g.*, sensors dropped in the atmosphere or flowing in the ocean).

To allow for a more systematic extension of our sensors with state variables we can assume that each sensor has a heap H where the values of these variables are stored: $[H, P, M]_b^{p,r}$. The model chosen for this heap is orthogonal to our sensor calculus and for this discussion we assume that we enrich the values v of the language with a set of *keys*, ranged over by k. Our heap may thus be defined as a map H from $keys$ into $values$. Intuitively, we can think of it as an associative memory with the usual built-in operations **put**, **get**, **lookup**, and **hash**. Programs running in the sensors may share state by exchanging keys. We assume also that these operations are atomic and thus no race conditions can arise.

With this basic model for a heap we can re-implement the *Ping* example from Section III-A with *scoped flooding* thus eliminating echos by software. We do this by associating a unique key to each remote procedure call broadcast to the network. This key is created through the built-in **hash** function that takes as arguments the position p and the battery b of the sensor. Each sensor, after receiving a call to ping, propagates the call to its neighborhood and generates a new key to send, with its position and battery charge, in a forward call. Then, it stores the key in its heap to avoid forwarding its own forward call. On the other hand, each time a sensor receives a call to forward, it checks whether it has the key associated to the call in its heap. If so, it does nothing. If not, it forwards the call and stores the key in the heap, to avoid future re-transmission.

```
MSensor(p, b) = {
  ping       = ()        net.ping[];
                         let k = hash[p,b] in net.forward[p, b, k];
                         put[k,_]
  forward    = (x, y, k) if (!lookup[k]) then
                            net.forward[x, y, k];
                            put[k,_]
}
MSink(p, b)    = {...}
[net.ping[], MSink]^{p,r}_b |
[idle, MSensor(p_1,b_1)]^{p_1,r_1}_{b_1} | ... | [idle, MSensor(p_n,b_n)]^{p_n,r_n}_{b_n}
```

b) Events: Another characteristic of sensors is their *modus operandi*. Some sensors sample the field as a result of instructions implemented in the software that controls them. Such is the case with CSN

sensors. The programmer is responsible for controlling the sensing activity of the sensor network. It is of course possible for sensor nodes to be activated in different ways. For example, some may have their sensing routines implemented at hardware or operating system level and thus not directly controllable by the programmer. Such classes of sensor nodes tipically sample the field periodically and are activated when a given condition arises (*e.g.*, a temperature above or below a given threshold, the detection of CO_2 above a given threshold, the detection of a strong source of infrared light). The way in which certain environmental conditions or *events* can activate the sensor is by triggering the execution of a handler procedure that processes the event. Support for this kind of event-driven sensors in CSN could be achieved by assuming that each sensor has a builtin handler procedure, say handle, for such events. The handler procedure, when activated, receives the value of the field that triggered the event. Note that, from the point of view of the sensor, the occurrence of such an event is equivalent to the deployment of a method invocation **this**.handle[v] in its processing core, where v is the field value associated with the event. The sensor has no control over this deployment, but may be programmed to react in different ways to these calls, by providing adequate implementations of the handle routine. The events could be included in the semantics given in Section II-C with the following rule:

$$[P, M]_b^{p,r} \rightarrow_F [\textbf{this}.\mathsf{handle}\,[F(p)] \mid P, M]_b^{p,r} \qquad \text{(R-EVENT)}$$

As in the case of the builtin method for code deployment, the handler could be programmed to change the behavior of the network in the presence of events. One could envision the default handler as handle = (x) **idle**, which ignores all events. Then, we could change this default behavior so that an event triggers an alarm that gets sent to the sink. A possible implementation of such a dynamic re-programming of the network default handlers can be seen in the code below.

```
MSensor(p)  = { handle  = (x) idle }
MSink(p)    = { handle  = (x) idle }
[net.deploy[{
     handle = (x)   net.alarm[p,x]
     alarm  = (x,y) net.alarm[x,y]
 }];
 install {alarm  = (x,y) sing_bell[x,y]},
 MSink(p)] b^{p,r} |
[idle, MSensor(p1)] b1^{p1,r1}  |  ...  |  [idle, MSensor(pn)] bn^{pn,rn}
```

where the default implementation of the handle procedure is superseded by one that eventualy triggers an alarm in the sink.

More complex behavior could be modeled for sensors that take multiple readings, with a handler associated with each event.

c) Security: Finally, another issue that is of outmost importance in the management of sensor networks is security. It is important to note that many potential applications of sensor networks are in high risk situations. Examples may be the monitorization of ecological disaster areas, volcanic or sismic activity, and radiation levels in contaminated areas. Secure access to data is fundamental to establish its credibility and for correctly assessing risks in the management of such episodes. In CSN we have not taken security issues into consideration. This was not our goal at this time. However, one feature of the calculus may provide interesting solutions for the future. In fact, in CSN, all computation within a sensor results from an invocation of methods in the modules of a sensor, either originating in the network or from within the sensor. In a sense the modules M of the sensor work as a firewall that can be used to control incomming messages and implement security protocols. Thus, all remote method invocations and software updates might first be validated locally with methods of the sensor's modules and only then the actions would be performed. The idea of equipping sensors, or in general *domains*, with some kind of membrane that filters all the interactions with the surrounding network has been explored, for instance, in [6], in the M-calculus [25], in the Kell calculus [26], in the Brane calculi [7], in Miko [18], and in [12]. One possible development is to incorporate some features of the *membrane model* into CSN. The current formulation of the calculus also assumes that all methods in the module M of a sensor $[P, M]_b^{p,r}$ are visible from the network. It is possible to implement an access policy to methods in such a way that some methods are private to the sensor, *i.e.*, can only be invoked from within the sensor. This allows, for example, the complete encapsulation of the state of the sensor.

V. Conclusions and Future Work

Aiming at providing large-scale sensor networks with a rigorous and adequate programming model (upon which operating systems and high-level programming languages can be built), we presented CSN — a Calculus for Sensor Networks, developed specifically for this class of distributed systems.

After identifying the necessary sensing, processing, and wireless broadcasting features of the calculus, we opted to base our work on a top-layer abstraction of physical and link layer communication issues (in contrast with previous work on wireless network calculi [19, 23]), thus focusing on the system

requirements for programming network-wide applications. This approach resulted in the CSN syntax and semantics, whose expressiveness we illustrated through a series of implementations of typical operations in sensor networks. Also included was a detailed discussion of possible extensions to CSN to account for other important properties of sensors such as state, sampling strategies, and security.

As part of our ongoing efforts, we are currently using CSN to establish a mathematical framework for reasoning about sensor networks. One major objective of this work consists in providing formal proofs of correctness for data gathering protocols that are commonly used in current sensor networks and whose performance and reliability has so far only been evaluated through computer simulations and ad-hoc experiments.

From a more practical point of view, the focus will be set on the development of a prototype implementation of CSN. This prototype will be used to emulate the behavior of sensor networks by software and, ultimately, to port the programming model to a natural development architecture for sensor network applications.

ACKNOWLEDGEMENTS

The authors gratefully acknowledge insightful discussions with Gerhard Maierbacher (Departamento de Ciência de Computadores, Faculdade de Ciências, Universidade do Porto).

REFERENCES

[1] The TinyOS Documentation Project. Available at `http://www.tinyos.org`.

[2] M. Abadi and L. Cardelli. An Imperative Object Calculus. In *TAPSOFT '95: Theory and Practice of Software Development*, number 915 in LNCS, pages 471–485. Springer-Verlag, 1995.

[3] I. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. A survey on sensor networks. *IEEE Communications Magazine*, 40(8):102–114, 2002.

[4] J. Barros and S. D. Servetto. Network information flow with correlated sources. IEEE Transactions on Information Theory, Vol. 52, No. 1, pp. 155-170, January 2006.

[5] G. Boudol. Asynchrony and the pi-calculus. Technical Report 1702, INRIA, Institut National de Recherche en Informatique et en Automatique, 1992.

[6] G. Boudol. A generic membrane model. In *Global Computing Workshop*, volume 3267 of *LNCS*, pages 208–222. Springer-Verlag, 2005.

[7] L. Cardelli. Brane calculi: Interactions of biological membranes. In *Proceedings of CMSB'04*, volume 3082 of *LNCS*, pages 257–280. Springer-Verlag, 2004.

[8] D. E. Culler and H. Mulder. Smart sensors to network the world. *Scientific American*, 2004.

[9] C.-L. Fok, G.-C. Roman, and C. Lu. Rapid Development and Flexible Deployment of Adaptive Wireless Sensor Network Applications. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'05)*, pages 653–662. IEEE, June 2005.

[10] C. Fournet and G. Gonthier. The Reflexive Chemical Abstract Machine and the Join-Calculus. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)*, pages 372–385. ACM, 1996.

[11] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC Language: A Holistic Approach to Network Embedded Systems. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*., 2003.

[12] D. Gorla, M. Hennessy, and V. Sassone. Security policies as membranes in systems for global computing. In *Proceedings of FGUC'04*, ENTCS. Elsevier Science, 2004.

[13] K. Honda and M. Tokoro. An object calculus for asynchronous communication. In *Proceedings of the ECOOP '91 European Conference on Object-oriented Programming*, LNCS 512, pages 133–147. Springer-Verlag, 1991.

[14] Z. Hu and B. Li. On the fundamental capacity and lifetime limits of energy-constrained wireless sensor networks. In *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2004)*, pages 38–47, Toronto, Canada, 2004.

[15] J. W. Hui and D. Culler. The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 81–94. ACM Press, 2004.

[16] R. Pugliese L. Bettini, R. De Nicola. X-Klaim and Klava: Programming Mobile Code. *TOSCA 2001, Electronic Notes on Theoretical Computer Science, Elsevier*, 62, 2001.

[17] P. Levis and D. Culler. Maté: A tiny virtual machine for sensor networks. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, 2002.

[18] F. Martins, L. Salvador, V. Vasconcelos, and L. Lopes. Miko: Mikado koncurrent objects. Technical Report 05081, Dagstuhl Seminar, 2005.

[19] N. Mezzetti and D. Sangiorgi. Towards a Calculus for Wireless Systems. In *Proc. MFPS '06*, volume 158 of *ENTCS*, pages 331–354. Elsevier, 2006.

[20] R. Milner. *A Calculus of Communicating Systems*, volume 92. Springer-Verlag, 1980.

[21] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, (Parts I and II). *Information and Computation*, 100:1–77, 1992.

[22] K. Ostrovský, K. V. S. Prasad, and W. Taha. Towards a Primitive Higher Order Calculus of Broadcasting Systems. In *PPDP'02, International Conference on Principles and Practice of Declarative Programming*, 2002.

[23] K. V. S. Prasad. A Calculus of Broadcasting Systems. In *TAPSOFT, Volume 1*, pages 338–358, 1991.

[24] A. Scaglione and S. D. Servetto. On the Interdependence of Routing and Data Compression in Multi-Hop Sensor Networks. In *Proc. ACM MobiCom*, Atlanta, GA, 2002.

[25] A. Schmitt and J.-B. Stefani. The M-calculus: a higher-order distributed process calculus. In *Proceedings of POPL'03*, pages 50–61. ACM Press, 2003.

[26] J.-B. Stefani. A calculus of Kells. In *Proceedings of FGC'03*, volume 85(1). Elsevier Science, 2003.

[27] V. Vasconcelos, L. Lopes, and F. Silva. Distribution and Mobility with Lexical Scoping in Process Calculi. In *Workshop on High Level Programming Languages (HLCL'98)*, volume 16(3) of *ENTCS*, pages 19–34. Elsevier Science, 1998.

[28] P. T. Wojciechowski and P. Sewell. Nomadic Pict: Language and Infrastructure Design for Mobile Agents. *IEEE Concurrency*, 8(2):42–52, /2000.

A Calculus for Sensor Networks

Miguel S. Silva*, Francisco Martins†, Luís Lopes*, and João Barros*

*Departamento de Ciência de Computadores & LIACC
Faculdade de Ciências da Universidade do Porto, Portugal.
†Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa, Portugal.

arXiv:cs/0612093v1 [cs.DC] 19 Dec 2006

Abstract

We consider the problem of providing a rigorous model for programming wireless sensor networks. Assuming that collisions, packet losses, and errors are dealt with at the lower layers of the protocol stack, we propose a Calculus for Sensor Networks (CSN) that captures the main abstractions for programming applications for this class of devices. Besides providing the syntax and semantics for the calculus, we show its expressiveness by providing implementations for several examples of typical operations on sensor networks. Also included is a detailed discussion of possible extensions to CSN that enable the modeling of other important features of these networks such as sensor state, sampling strategies, and network security.

keywords: Sensor Networks, Ad-Hoc Networks, Ubiquitous Computing, Process-Calculi, Programming Languages.

I. Introduction

A. The Sensor Network Challenge

Sensor networks, made of tiny, low-cost devices capable of sensing the physical world and communicating over radio links [**?**], are significantly different from other wireless networks: (a) the design of a sensor network is strongly driven by its particular application, (b) sensor nodes are highly constrained in terms of power consumption and computational resources (CPU, memory), and (c) large-scale sensor applications require self-configuration and distributed software updates without human intervention. Previous work on fundamental aspects of wireless sensor networks has mostly focused on communication-oriented models, in which the sensor nodes are assumed to store and process the data, coordinate their transmissions, organize the routing of messages within the network, and relay the data to a remote receiver (see *e.g.* [**?**, **?**, **?**] and references therein). Although some of these models provide useful insights (*e.g.* into the

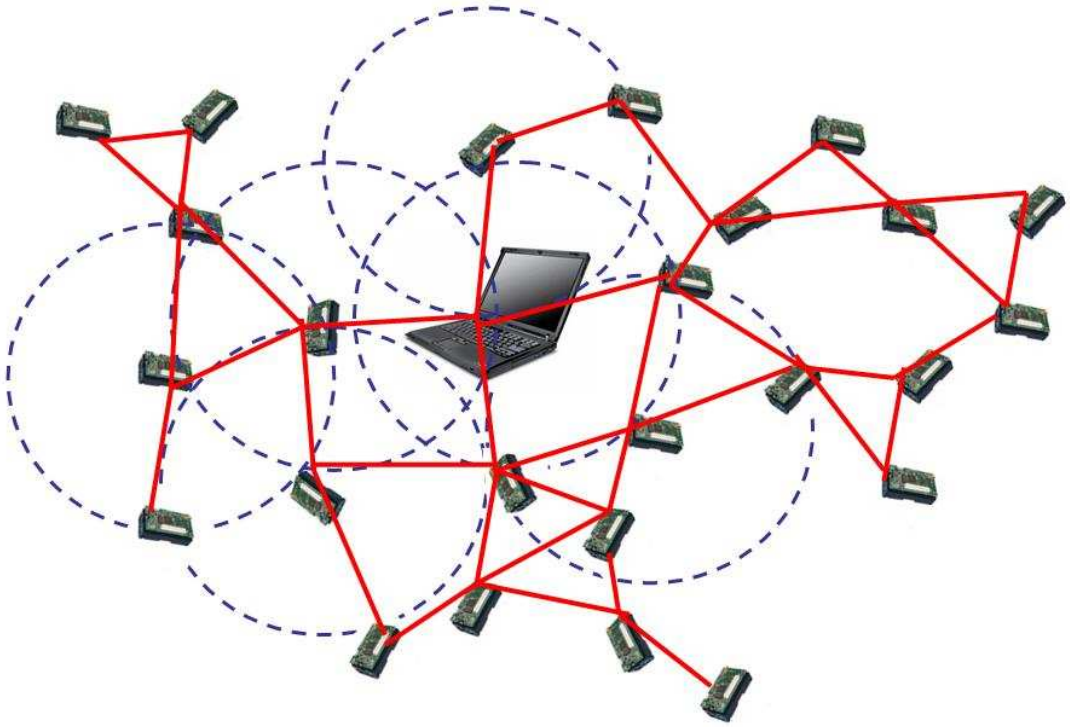

Fig. 1. A wireless sensor network is a collection of small devices that, once deployed on a target area, organize themselves in an ad-hoc network, collect measurements of a physical process and transmit the data over the wireless medium to a data fusion center for further processing.

connectivity characteristics or the overall power efficiency of sensor networks) there is a strong need for formal methods that capture the inherent processing and memory constraints, and illuminate the massively parallel nature of the sensor nodes' in-network processing. If well adapted to the specific characteristics of sensor networks, a formalism of this kind, specifically a process calculus, is likely to have a strong impact on the design of operating systems, communication protocols, and programming languages for this class of distributed systems.

In terms of hardware development, the state-of-the-art is well represented by a class of multi-purpose sensor nodes called *motes*[1] [**?**], which were originally developed at UC Berkeley and are being deployed and tested by several research groups and start-up companies. In most of the currently available implementations, the sensor nodes are controlled by module-based operating systems such as TinyOS [**?**] and programming languages like nesC [**?**] or TinyScript/Maté [**?**]. In our view, the programming models

[1]Trademark of Crossbow Technology, Inc.

underlying most of these tools have one or more of the following drawbacks:

1) they do not provide a rigorous model (or a calculus) of the sensor network at the programming level, which would allow for a formal verification of the correctness of programs, among other useful analysis;
2) they do not provide a global vision of a sensor network application, as a specific distributed application, making it less intuitive and error prone for programmers;
3) they require the programs to be installed on each sensor individually, something unrealistic for large sensor networks;
4) they do not allow for dynamic re-programming of the network.

Recent middleware developments such as Deluge [**?**] and Agilla [**?**] address a few of these drawbacks by providing higher level programming abstractions on top of TinyOS, including massive code deployment. Nevertheless, we are still far from a comprehensive programming solution with strong formal support and analytical capabilities.

The previous observation motivates us to design a sensor network programming model from scratch. Beyond meeting the challenges of network-wide programming and code deployment, the model should be capable of producing quantitative information on the amount of resources required by sensor network programs and protocols, and also of providing the necessary tools to prove their correctness.

B. Related Work

Given the distributed and concurrent nature of sensor network operations, we build our sensor network calculus on thirty years of experience gathered by concurrency theorists and programming language designers in pursuit of an adequate formalism and theory for concurrent systems. The first steps towards this goal were given by Milner [**?**] with the development of CCS (Calculus of Communicating Systems). CCS describes computations in which concurrent processes may interact through simple synchronization, without otherwise exchanging information. Allowing processes to exchange resources (*e.g.*, links, memory references, sockets, code), besides synchronizing, considerably increases the expressive power of the formal systems. Such systems, known as process-calculi, are able to model the mobility patterns of the resources and thus constitute valuable tools to reason about concurrent, distributed systems.

The first such system, built on Milner's work, was the π-calculus [**?**]. Later developments of this initial proposal allowed for further simplification an provided an asynchronous form of the calculus [**?**, **?**]. Since then, several calculi have been proposed to model concurrent distributed systems and for many there are

prototype implementations of programming languages and run-time systems (*e.g.* Join [**?**], TyCO [**?**], X-Klaim [**?**], and Nomadic Pict [**?**]).

Previous work by Prasad [**?**] established the first process calculus approach to modeling broadcast based systems. Later work by Ostrovský, Prasad, and Taha [**?**] established the basis for a higher-order calculus for broadcasting systems. The focus of this line of work lies in the protocol layer of the networks, trying to establish an operational semantics and associated theory that allows assertions to be made about the networks. More recently, Mezzetti and Sangiorgi [**?**] discuss the use of process calculi to model wireless systems, again focusing on the details of the lower layers of the protocol stack (*e.g.* collision avoidance) and establishing an operational semantics for the networks.

C. Our Contributions

Our main contribution is a sensor network programming model based on a process calculus, which we name Calculus of Sensor Networks (CSN). Our calculus offers the following features that are specifically tailored for sensor networks:

- *Top-Level Approach*: CSN focuses on programming and managing sensor networks and so it assumes that collisions, losses, and errors have been dealt with at the lower layers of the protocol stack and system architecture (this distinguishes CSN from the generic wireless network calculus presented in [**?**]);
- *Scalability*: CSN offers the means to provide the sensor nodes with self-update and self-configuration abilities, thus meeting the challenges of programming and managing a large-scale sensor network;
- *Broadcast Communication*: instead of the peer-to-peer (unicast) communication of typical process calculi, CSN captures the properties of broadcast communication as favored by sensor networks (with strong impact on their energy consumption);
- *Ad-hoc Topology*: network topology is not required to be programmed in the processes, which would be unrealistic in the case of sensor networks;
- *Communication Constraints*: due to the power limitations of their wireless interface, the sensor nodes can only communicate with their direct neighbors in the network and thus the notion of neighborhood of a sensor node, *i.e.* the set of sensor nodes within its communication range, is introduced directly in the calculus;
- *Memory and Processing Constraints*: the typical limitations of sensor networks in terms of memory and processing capabilities are captured by explicitly modeling the internal processing (or the *intelligence*) of individual sensors;

- *Local Sensing*: naturally, the sensors are only able to pick up local measurements of their environment and thus have geographically limited sensitivity.

To provide these features, we devise CSN as a two-layer calculus, offering abstractions for data acquisition, communication, and processing. The top layer is formed by a network of sensor nodes immersed in a scalar or vector field (representing the physical process captured by the sensor nodes). The sensor nodes are assumed to be running in parallel. Each sensor node is composed of a collection of labeled methods, which we call a module, and that represents the code that can be executed in the device. A process is executed in the sensor node as a result of a remote procedure call on a module by some other sensor or, seen from the point of view of the callee, as a result of the reception of a message. Sensor nodes are multithreaded and may share state, for example, in a tuple-space. Finally, by adding the notions of position and range, we are able to capture the nature of broadcast communication and the geographical limits of the sensor network applications.

The remainder of this paper is structured as follows. The next section describes the syntax and semantics of the CSN calculus. Section III presents several examples of functionalities that can be implemented using CSN and that are commonly required in sensor networks. In Section IV we discuss some design options we made and how we can extend CSN to model other aspects of sensor networks. Finally, Section V presents some conclusions and directions for future work.

II. The Calculus

This section addresses the syntax and the semantics of the Calculus for Sensor Networks. For simplicity, in the remainder of the paper we will refer to a sensor node or a sensor device in a network as a *sensor*. The syntax is provided by the grammar in Figure 2, and the operational semantics is given by the reduction relation depicted in Figures 3 and 4.

A. Syntax

Let $\vec{\alpha}$ denote a possible empty sequence $\alpha_1 \dots \alpha_n$ of elements of the syntactic category α. Assume a countable set of *labels*, ranged over by letter l, used to name methods within modules, and a countable set of *variables*, disjoint from the set of labels and ranged over by letter x. Variables stand for communicated values (*e.g.* battery capacity, position, field measures, modules) in a given program context.

The syntax for CNS is found in Figure 2. We explain the syntactic constructs along with their informal, intuitive semantics. Refer to the next section for a precise semantics of the calculus.

N ::=	*Network*	P ::=	*Programs*
S, F	sensors and field	**idle**	idle
		$\mid P \mid P$	parallel composition
S ::=	*Sensors*	$\mid P \,;\, P$	sequential composition
off	termination	$\mid t.v[\vec{v}]$	method invocation
$\mid N \mid N$	composition	$\mid$ **install** v	module update
$\mid [P, M]_b^{p,r}$	sensor	$\mid$ **sense** $(\vec{x})$ **in** P	field sensing
$\mid [P, M]_b^{p,r}\{S\}$	broadcast sensor	$\mid$ **if** v **then** P **else** P	conditional execution
M ::=	*Modules*	v ::=	*Values*
$\{l_i = (\vec{x}_i)\, P_i\}_{i \in I}$	method collection	x	variable
		$\mid m$	field measure
t ::=	*Targets*	$\mid p$	position
net	broadcast	$\mid b$	battery capacity
$\mid$ **this**	local	$\mid M$	module

Fig. 2. The syntax of CSN.

Networks N denote the composition of sensor networks S with a (scalar or vector) field F. A field is a set of pairs (position, measure) describing the distribution of some physical quantity (*e.g.* temperature, pressure, humidity) in space. The position is given in some coordinate system. Sensors can measure the intensity of the field in their respective positions.

Sensor networks S are flat, unstructured collections of sensors combined using the parallel composition operator.

A sensor $[P, M]_b^{p,r}$ represents an abstraction of a physical sensing device and is parametric in its position p, describing the location of the sensor in some coordinate system; its transmission range specified by the radius r of a circle centered at position p; and its battery capacity b. The position of the sensors may

vary with time if the sensor is mobile in some way. The transmission range, on the other hand, usually remains constant over time. A sensor with the battery exhausted is designated by **off**.

Inside a sensor there exists a running program P and a module M. A module is a collection of methods defined as $l = (\vec{x})P$ that the sensor makes available for internal and for external usage. A method is identified by label l and defined by an abstraction $(\vec{x})P$: a program P with parameters $\vec{x}$. Method names are pairwise distinct within a module. Mutually recursive method definitions make it possible to represent infinite behavior. Intuitively, the collection of methods of a sensor may be interpreted as the function calls of some tiny operating system installed in the sensor.

Communication in the sensor network only happens via broadcasting values from one sensor to its neighborhood: the sensors inside a circle centered at position p (the position of the sensor) with radius r. A broadcast sensor $[P, M]_b^{p,r}\{S\}$ stands for a sensor during the broadcast phase, having already communicated with sensors S. While broadcasting, it is fundamental to keep track of the sensors engaged in communication so far, thus preventing the delivery of the same message to the same sensor during one broadcasting operation. Target sensors are collected in the *bag* of the sensor emitting the message. Upon finishing the broadcast the bag is emptied out, and the (target) sensors are released into the network. This construct is a run-time construct and is available to the programmer.

Programs are ranged over by P. The **idle** program denotes a terminated thread. Method invocation, $t.v[\vec{v}]$, selects a method v (with arguments $\vec{v}$) either in the local module or broadcasts the request to the neighborhood sensors, depending whether t is the keyword **this** or the keyword **net**, respectively. Program **sense** $(\vec{x})$ **in** P reads a measure from the surrounding field and binds it to $\vec{x}$ within P. Installing or replacing methods in the sensor's module is performed using the construct **install** v. The calculus also offers a standard form of branching through the **if** v **then** P **else** P construct.

Programs P and Q may be combined in sequence, $P\ ;\ Q$, or in parallel, $P \,|\, Q$. The sequential composition $P\ ;\ Q$ designates a program that first executes P and then proceeds with the execution of Q. In contrast, $P \,|\, Q$ represents the simultaneous execution of P and Q.

Values are the data exchanged between sensors and comprise field measures m, positions p, battery capacities b, and modules M. Notice that this is not a higher-order calculus: communicating a module means the ability to transfer its *code* to, to retransmit it from, or to install it in a remote sensor.

B. Examples

Our first example illustrates a network of sensors that sample the field and broadcast the measured values to a special node known as the *sink*. The sink node may be no different from the other sensors

in the network, except that it usually possesses a distinct software module that allows it to collect and process the values broadcasted in the network. The behavior we want to program is the following. The sink issues a request to the network to sample the field; upon reception of the request each sensor samples the field at its position and broadcasts the measured value back to the sink; the sink receives and processes the values. An extended version of this example may be found in Section III-B.

The code for the modules of the sensors, MSensor(p, r), and for the sink, MSink(p, r), is given below. Both modules are parametric in the position and in the broadcasting range of each sensor.

As for the module equipping the sensors, it has a method sample that, when invoked, propagates the call to its neighborhood (**net**.sample[];), samples the field (**sense** x **in** ...) and forwards the value to the network (... **net**.forward[p,x]). Notice that each sensor propagates the original request from the sink. This is required since in general most of the sensors in the network will be out of broadcasting range from the sink. Therefore each sensor echos the request, hopefully covering all the network. Message forwarding will be a recurrent pattern found in our examples. Another method of the sensors' module is forward that simply forwards the values from other sensors through the network.

The module for the sink contains a different implementation of the forward method, since the sink will gather the values sent by the sensors and will log them. Here we leave unspecified the processing done by the log_position_and_value program.

The network starts-up with all sensors **idle**, except for the sink that requests a sampling (**net**.sample[]).

```
MSensor(p,r) = { sample  = () net.sample[]; sense x in net.forward[p,x]
               forward = (x,y) net.forward[x,y] }
MSink(p,r)   = { forward = (x,y) log_position_and_value[x,y] }

[net.sample[], MSink(p,r)] b^{p,r} |
[idle, MSensor(p1,r1)] b1^{p1,r1} | ... | [idle, MSensor(pn,rn)] bn^{pn,rn}
```

The next example illustrates the broadcast, the deployment, and the installation of code. The example runs as follows. The sink node deploys some module in the network (**net**.deploy[M]) and then seals the sensors (**net**.seal[]), henceforth preventing any dynamic re-programming of the network. An extended version of the current example may be found in Section III-E.

The code for the modules of the sensors and of the sink is given below. The module M is the one we wish to deploy to the network. It carries the method seal that forwards the call to the network and installs a new version of deploy that does nothing when executed.

$$P_1 \mid P_2 \equiv P_2 \mid P_1, \qquad P \mid \textbf{idle} \equiv P, \qquad P_1 \mid (P_2 \mid P_3) \equiv (P_1 \mid P_2) \mid P_3 \qquad \text{(S-MONOID-PROGRAM)}$$

$$S_1 \mid S_2 \equiv S_2 \mid S_1, \qquad S \mid \textbf{off} \equiv S, \qquad S_1 \mid (S_2 \mid S_3) \equiv (S_1 \mid S_2) \mid S_3 \qquad \text{(S-MONOID-SENSOR)}$$

$$\textbf{idle} \ ; \ P \equiv P \qquad \frac{P_1 \equiv P_2}{[P_1, M]_b^{p,r} \equiv [P_2, M]_b^{p,r}} \qquad \text{(S-IDLE-SEQ, S-PROGRAM-STRU)}$$

$$[P, M]_b^{p,r} \equiv [P, M]_b^{p,r}\{\textbf{off}\} \qquad \frac{b < \max(\mathsf{c_{in}}, \mathsf{c_{out}})}{[P, M]_b^{p,r} \equiv \textbf{off}} \qquad \text{(S-BROADCAST, S-BAT-EXHAUSTED)}$$

Fig. 3. Structural congruence for processes and sensors.

```
MSensor(p,r) = { deploy = (x) net.deploy[x]; install x }
MSink(p,r)   = {}
M            = { seal   = ()  net.seal[]; install { deploy = () idle } }

[net.deploy[M]; net.seal[],MSink(p,r)]^{p,r}_b |
[idle,MSensor(p1,r1)]^{p1,r1}_b1 | ... | [idle,MSensor(pn,rn)]^{pn,rn}_bn
```

C. Semantics

The calculus has two name bindings: field sensing and method definitions. The displayed occurrence of name x_i is a *binding* with *scope* P both in **sense** $(x_1, \ldots, x_i, \ldots, x_n)$ **in** P and in $l = (x_1, \ldots, x_i, \ldots, x_n)\, P$. An occurrence of a name is *free* if it is not in the scope of a binding. Otherwise, the occurrence of the name is *bound*. The set of free names of a sensor S is referred as $\mathrm{fn}(S)$.

Following Milner [**?**] we present the reduction relation with the help of a structural congruence relation. The structural congruence relation $\equiv$, depicted in Figure 3, allows for the manipulation of term structure, adjusting sub-terms to reduce. The relation is defined as the smallest congruence relation on sensors (and programs) closed under the rules given in Figure 3.

The parallel composition operators for programs and for sensors are taken to be commutative and associative with **idle** and **off** as their neutral elements, respectively (*vide* Rules S-MONOID-PROGRAM and S-MONOID-SENSOR). Rule S-IDLE-SEQ asserts that **idle** is also neutral with respect to sequential composition of programs. Rule S-PROGRAM-STRU incorporates structural congruence for programs into sensors. When a sensor is broadcasting a message it uses a bag to collect the sensors as they become engaged in communication. Rule S-BROADCAST allows for a sensor to start the broadcasting operation.

$$\frac{M(l_i) = (\vec{x}_i)P_i \qquad b \geq \mathsf{c_{in}}}{[\mathbf{this}\,.l_i[\vec{v}]\ ;\ P_1 \mid P_2, M]_b^{p,r} \rightarrow_F [P_i[\vec{v}/\vec{x}_i]\ ;\ P_1 \mid P_2, M]_{b-\mathsf{c_{in}}}^{p,r}} \quad \text{(R-METHOD)}$$

$$\frac{l_i \notin \mathrm{dom}(M)}{[\mathbf{this}\,.l_i[\vec{v}]\ ;\ P_1 \mid P_2, M]_b^{p,r} \rightarrow_F [\mathbf{this}\,.l_i[\vec{v}]\ ;\ P_1 \mid P_2, M]_b^{p,r}} \quad \text{(R-NO-METHOD)}$$

$$\frac{\mathrm{d}(p,p') < r \qquad b \geq \mathsf{c_{out}}}{\begin{array}{l}[\mathbf{net}\,.l_i[\vec{v}]\ ;\ P_1 \mid P_2, M]_b^{p,r}\{S\} \mid [P', M']_{b'}^{p',r'} \rightarrow_F \\ \qquad [\mathbf{net}\,.l_i[\vec{v}]\ ;\ P_1 \mid P_2, M]_b^{p,r}\{S \mid [P' \mid \mathbf{this}\,.l_i[\vec{v}], M']_{b'}^{p',r'}\}\end{array}} \quad \text{(R-BROADCAST)}$$

$$[\mathbf{net}\,.l_i[\vec{v}]\ ;\ P_1 \mid P_2, M]_b^{p,r}\{S\} \rightarrow_F [P_1 \mid P_2, M]_{b-\mathsf{c_{out}}}^{p,r} \mid S \quad \text{(R-RELEASE)}$$

$$\frac{b \geq \mathsf{c_{in}}}{[\mathbf{install}\ M'\ ;\ P_1 \mid P_2, M]_b^{p,r} \rightarrow_F [P_1 \mid P_2, M + M']_{b-\mathsf{c_{in}}}^{p,r}} \quad \text{(R-INSTALL)}$$

$$\frac{b \geq \mathsf{c_{in}}}{[\mathbf{sense}\ (\vec{x})\ \mathbf{in}\ P\ ;\ P_1 \mid P_2, M]_b^{p,r} \rightarrow_F [P[F(p)/\vec{x}]\ ;\ P_1 \mid P_2, M]_{b-\mathsf{c_{in}}}^{p,r}} \quad \text{(R-SENSE)}$$

$$\frac{S_1 \rightarrow_F S_2}{S \mid S_1 \rightarrow_F S \mid S_2} \qquad \frac{S_1 \equiv S_2 \qquad S_2 \rightarrow_F S_3 \qquad S_3 \equiv S_4}{S_1 \rightarrow_F S_4} \quad \text{(R-PARALLEL, R-STRUCTURAL)}$$

$$\frac{S \rightarrow_F S'}{S, F \rightarrow S', F} \quad \text{(R-NETWORK)}$$

Fig. 4. Reduction semantics for processes and networks.

A terminated sensor is a sensor with insufficient battery capacity for performing an internal or an external reduction step (*vide* Rule S-BAT-EXHAUSTED).

The reduction relation on networks, notation $S, F \rightarrow S', F$, describes how sensors S can evolve (reduce) to sensors S', sensing the field F. The reduction is defined on top of a reduction relation for sensors, notation $S \rightarrow_F S'$, inductively defined by the rules in Figure 4. The reduction for sensors is parametric on field F and on two constants $\mathsf{c_{in}}$ and $\mathsf{c_{out}}$ that represent the amount of energy consumed when performing internal computation steps ($\mathsf{c_{in}}$) and when broadcasting messages ($\mathsf{c_{out}}$).

Computation inside sensors proceeds by invoking a method (either local—Rules R-METHOD and R-NO-METHOD—or remote—Rules R-BROADCAST and R-RELEASE), by sensing values (Rule R-SENSE), and by updating the method collection of the sensor (Rule R-INSTALL).

The invocation of a local method l_i with arguments $\vec{v}$ evolves differently depending on whether or not the definition for l_i is part of the method collection of the sensor. Rule R-METHOD describes the invocation of a method from module M, defined as $M(l_i) = (\vec{x}_i)P_i$. The result is the program P_i where the values $\vec{v}$ are bound to the variables in $\vec{x}$. When the definition for l_i is not present in M, we have decided to actively wait for the definition (see Rule R-NO-METHOD). Usually invoking an undefined method causes a program to get *stuck*. Typed programming languages use a type system to ensure that there are no invocations to undefined methods, ruling out all other programs at compile time. At run-time, another possible choice would be to simply discard invocations to undefined methods. Our choice provides more resilient applications when coupled with the procedure for deploying code in a sensor network. We envision that if we invoke a method in the network after some code has been deployed (see Example III-D), there may be some sensors where the method invocation arrives before the deployed code. With the semantics we propose, the call actively waits for the code to be installed.

Sensors communicate with the network by broadcasting messages. A message consists of a remote method invocation on unspecified sensors in the neighborhood of the emitting sensor. In other words, the messages are not targeted to a particular sensor (there is no peer-to-peer communication). The neighborhood of a sensor is defined by its communication radius, but there is no guarantee that a message broadcasted by a given sensor arrives at all surrounding sensors. There might be, for instance, landscape obstacles that prevent two sensors, otherwise within range, from communicating with each other. Also, during a broadcast operation the message must only reach each neighborhood sensor once. Notice that we are not saying that the same message can not reach the same sensor multiple times. In fact it might, but as the result of the echoing of the message in subsequent broadcast operations. We model the broadcasting of messages in two stages. Rule R-BROADCAST invokes method l_i in the remote sensor, provided that the distance between the emitting and the receiving sensors is less that the transmission radius ($\mathrm{d}(p, p') < r$). The sensor receiving the message is put in the bag of the emitting sensor, thus preventing multiple deliveries of the same message while broadcasting. Observe that the rule does not enforce the interaction with all sensors in the neighborhood. Rule R-RELEASE finishes the broadcast by consuming the operation (**net** $.l_i[\vec{v}]$), and by emptying out the contents of the emitting sensor's bag. A broadcast operation starts with the application of Rule S-BROADCAST, proceeds with multiple (eventually none) applications of Rule R-BROADCAST (one for each target sensor), and terminates with the application of Rule R-RELEASE.

Installing module M' in a sensor with a module M, Rule R-INSTALL, amounts to add to M the methods in M' (absent in M), and to replace (in M) the methods common to both M and M'. Rigorously, the operation of installing module M' on top of M, denoted $M + M'$, may be defined as $M + M' =$

$(M \setminus M') \cup M'$. The $+$ operator is reminiscent of Abadi and Cardelli's operator for updating methods in their imperative object calculus [**?**].

A sensor senses the field in which it is immersed, Rule R-SENSE, by sampling the value of the field F in its position p and, continues the computation replacing this value for the bound variables $\vec{x}$ in program P.

Rule R-PARALLEL allows reduction to happen in networks of sensors and Rule R-STRUCTURAL brings structural congruence into the reduction relation.

D. The Operational Semantics Illustrated

To illustrate the operational semantics of CNS, we present the reduction steps for the examples discussed at the end of Section II-B. During reduction we suppress the side annotations when writing the sensors. Due to space constraints we consider a rather simple network with just the sink and another sensor.

[**net** . sample [] , MSink (p , r)] | [**idle** , MSensor (p_1 , r_1)]

We assume that the sensor is within range from the sink and vice-versa. This network may reduce as follows:

[**net**.sample[],MSink(p,r)] | [**idle**,MSensor(p_1,r_1)] $\equiv$ (S-BROADCAST)

[**net**.sample[],MSink(p,r)]{**off**} | [**idle**,MSensor(p_1,r_1)] $\rightarrow\equiv$

($\mathrm{d}(p, p_1) < r$, R-BROADCAST, S-MONOID-SENSOR)

[**net**.sample[],MSink(p,r)] {[**this**.sample[] || **idle**,MSensor(p_1,r_1)]} $\rightarrow\equiv$

(R-RELEASE, S-MONOID-PROGRAM)

[**idle**,MSink(p,r)] | [**this**.sample[],MSensor(p_1,r_1)] $\rightarrow$ (R-METHOD)

[**idle**,MSink(p,r)] | [**net**.sample[]; **sense** x **in net**.forward[p_1,x],MSensor(p_1,r_1)] $\equiv$ (S-BROADCAST)

[**idle**,MSink(p,r)] | [**net**.sample[]; **sense** x **in net**.forward[p_1,x],MSensor(p_1,r_1)]{**off**} $\rightarrow$ (R-RELEASE)

[**idle**,MSink(p,r)] | [**sense** x **in net**.forward[p_1,x],MSensor(p_1,r_1)] $\rightarrow$ (R-SENSE)

[**idle**,MSink(p,r)] | [**net**.forward[p_1,$F(p_1)$], MSensor(p_1,r_1)] $\equiv$ (S-MONOID-SENSOR)

[**net**.forward[p_1,$F(p_1)$], MSensor(p_1,r_1)] | [**idle**,MSink(p,r)] $\equiv$ (S-BROADCAST)

[**net**.forward[p_1,$F(p_1)$], MSensor(p_1,r_1)]{**off**} | [**idle**,MSink(p,r)] $\rightarrow\equiv$

($\mathrm{d}(p_1, p) < r_1$, R-BROADCAST, S-MONOID-SENSOR)

[**net**.forward[p_1,$F(p_1)$], MSensor(p_1,r_1)]{[**this**.forward[p_1,$F(p_1)$] || **idle**, MSink(p,r)]} $\rightarrow\equiv$

(R-RELEASE, S-MONOID-PROGRAM)

[**idle**, MSensor(p_1,r_1)] | [**this**.forward[p_1,$F(p_1)$], MSink(p,r)] $\rightarrow$ (R-METHOD)

[**idle**, MSensor(p_1,r_1)] | [log_position_and_value[p_1,$F(p_1)$], MSink(p,r)] $\equiv$ (S-MONOID-SENSOR)

[log_position_and_value[p_1,$F(p_1)$] | [**idle**, MSensor(p_1,r_1)], MSink(p,r)]

So, after these reduction steps the sink gets the field values from the sensor at position p_1 and logs them. The sensor at p_1 is idle waiting for further interaction.

Following we present the reduction step for our second (and last) example of Section II-B where we illustrate the broadcast, the deployment, and the installation of code. Again, due to space restrictions, we use a very simple network with just the sink and another sensor, both within reach of each other.

[**net**.deploy[M]; **net**.seal[],MSink(p,r)] | [**idle**,MSensor(p_1,r_1)]

This network may reduce as follows:

[**net**.deploy[M]; **net**.seal [], MSink(p,r)] | [**idle**,MSensor(p_1,r_1)] $\equiv$ (S-BROADCAST)

[**net**.deploy[M]; **net**.seal [], MSink(p,r)]{**off**} | [**idle**,MSensor(p_1,r_1)] $\rightarrow\equiv$
($\mathrm{d}(p, p_1) < r$, R-BROADCAST, S-MONOID-SENSOR)

[**net**.deploy[M]; **net**.seal [], MSink(p,r)] | {[**this**.deploy[M] || **idle**, MSensor(p_1,r_1)]} $\rightarrow\equiv$
(R-RELEASE, S-MONOID-PROGRAM)

[**net**.seal [], MSink(p,r)] | [**this**.deploy[M], MSensor(p_1,r_1)] $\rightarrow$ (R-METHOD)

[**net**.seal [], MSink(p,r)] | [**net**.deploy[M]; **install** M, MSensor(p_1,r_1)] $\equiv$ (S-BROADCAST)

[**net**.seal [], MSink(p,r)] | [**net**.deploy[M]; **install** M, MSensor(p_1,r_1)]{**off**} $\rightarrow\equiv$
(R-RELEASE, S-MONOID-SENSOR)

[**net**.seal [], MSink(p,r)] | [**install** M, MSensor(p_1,r_1)] $\rightarrow$ (R-INSTALL)

[**net**.seal [], MSink(p,r)] | [**idle**, MSensor(p_1,r_1)+M] $\equiv$ (S-BROADCAST)

[**net**.seal [], MSink(p,r)]{**off**} | [**idle**, MSensor(p_1,r_1)+M] $\rightarrow\equiv$
($\mathrm{d}(p, p_1) < r$, R-BROADCAST, S-MONOID-SENSOR)

[**net**.seal [], MSink(p,r)] {[**this**.seal [] || **idle**, MSensor(p_1,r_1)+M]} $\rightarrow\equiv$
(R-RELEASE, S-MONOID-PROGRAM)

[**idle**,MSink(p,r)] | [**this**.seal [], MSensor(p_1,r_1)+M] $\rightarrow$ (R-METHOD)

[**idle**,MSink(p,r)] | [**net**.seal []; **install** {deploy = () **idle**}, MSensor(p_1,r_1)+M] $\equiv$ (S-BROADCAST)

[**idle**,MSink(p,r)] | [**net**.seal []; **install** {deploy = () **idle**}, MSensor(p_1,r_1)+M]{**off**} $\rightarrow\equiv$
(R-RELEASE, S-MONOID-SENSOR)

[**idle**,MSink(p,r)] | [**install** {deploy = () **idle**}, MSensor(p_1,r_1)+M] $\rightarrow$ (R-INSTALL)

[**idle**,MSink(p,r)] | [**idle**, MSensor(p_1,r_1)+M+{deploy = () **idle**}]

After these reductions, the sink is idle after deploying the code to the sensor at p. The sensor at p is also idle, waiting for interaction, but with the code for the module M installed and with the deploy method disabled.

III. Programming Examples

In this section we present some examples, programmed in CSN, of typical operations performed on networks of sensors. Our goal is to show the expressiveness of the CSN calculus just presented and also to identify some other aspects of these networks that may be interesting to model. In the following examples, we denote as MSensor and MSink the modules installed in any of the anonymous sensors in the network and the modules installed in the sink, respectively. Note also that all sensors are assumed to have a builtin method, deploy, that is responsible for installing new modules. The intuition is that this method is part of the tiny operating system that allows sensors to react when first placed in the field. Finally, we assume in these small examples that the network layer supports *scoped flooding*. We shall see in the next section that this can be supported via software with the inclusion of state in sensors.

A. Ping

We start with a very simple ping program. Each sensor has a ping method that when invoked calls a method forward in the network with its position and battery charge as arguments. When the method forward is invoked by a sensor in the network, it just triggers another call to forward in the network. The sink has a distinct implementation of this method. Any incomming invocation logs the position and battery values given as arguments. So, the overall result of the call **net**.ping[] in the sink is that all reachable sensors in the network will, in principle, receive this call and will flood the network with their positions and battery charge values. These values eventually reach the sink and get logged.

```
MSensor(p,b) = {
  ping           = ()     net.forward[p,b]; net.ping[]
  forward        = (x,y)  net.forward[x,y]
}
MSink(p,b)   = {
  forward        = (x,y)  log_position_and_power[x,y]
}
[net.ping[], MSink(p,b)]^{p,r}_{b}  |
[idle, MSensor(p_1,b_1)]^{p_1,r_1}_{b_1}  |  ...  |  [idle, MSensor(p_n,b_n)]^{p_n,r_n}_{b_n}
```

B. Querying

This example shows how we can program a network with a sink that periodically queries the network for the readings of the sensors. Each sensor has a sample method that samples the field using the **sense**

construct and calls the method forward in the neighbourhood with its position and the value sampled as arguments. The call then queries the neighbourhood recursively with a replica of the original call. The original call is, of course, made from the sink, which has a method start_sample that calls the method sample in the network within a cycle. Note that, if the sink had a method named sample instead of start_sample, it might get a call to sample from elsewhere in the network that could interfere with the sampling control cycle.

```
MSensor(p)    = {
  sample          = ()      sense (x) in net.forward[p, x]; net.sample[]
  forward         = (x, y)  net.forward[x,y]
}
MSink(p)      = {
  start_sample    = ()      net.sample[]; this.start_sample[]
  forward         = (x, y)  log_position_and_value[x,y]
}
[this.start_sample[], MSink(p)] b^{p,r} |
[idle, MSensor(p_1)] b_1^{p_1,r_1} | ... | [idle, MSensor(p_n)] b_n^{p_n,r_n}
```

C. Polling

In this example the cycle of the sampling is done in each sensor, instead of in the sink, as in the previous example. The sink just invokes the method start_sample once. This method propagates the call through the network and invokes sample, for each sensor. This method samples the field, within a cycle, and forwards the result to the network. This implementation requires less broadcasts than the previous one as the sink only has to call start_sample on the network once. On the other hand, it increases the amount of processing per sensor.

```
MSensor(p)    = {
  start_sample    = ()      net.start_sample[]; this.sample[]
  sample          = ()      sense (x) in net.forward[p, x]; this.sample[]
  forward         = (x, y)  net.forward[x, y]
}
MSink(p)      = {
  forward         = (x, y)  log_position_and_value[x,y]
}
[net.start_example[], MSink(p)] b^{p,r} |
```

```
[idle, MSensor(p1)]_b1^p1,r1 | ... | [idle, MSensor(pn)]_bn^pn,rn
```

D. Code deployment

The above examples assume we have some means of deploying the code to the sensors. In this example we address this problem and show how it can be programmed in CSN. The code we wish to deploy and execute is the same as the one in the previous example. To achieve this goal, the sink first calls the deploy method on the network to install the new module with the methods start_sample, sample and forward as above. This call recursively deploys the code to the sensors in the network. The sink then calls start_sample to start the sampling, again as above, and waits for the forwarded results on the method forward.

```
MSensor(p)    = {
  deploy          = (x)     install x; net.deploy[x]
}
MSink(p)      = {
  forward         = (x,y)   log_position_and_value[x,y]
}
[net.deploy[{
  start_sample    = ()      net.start_sample[]; this.sample[]
  sample          = ()      sense (x) in net.forward[p, x]; this.sample[]
  forward         = (x, y)  net.forward[x, y]
 }];
 net.start_sample[], MSink(p)]_b^p,r |
[idle, MSensor(p1)]_b1^p1,r1 | ... | [idle, MSensor(pn)]_bn^pn,rn
```

A refined version of this code, one that avoids the start_sample method completely, can be programmed. Here, we deploy the code for all sensors by sending methods sample and forward to all the sensors in the network by invoking deploy. Once deployed, the code is activated with a call to sample in the sink, instead of using the start_sample method as above.

```
MSensor(p)    = {
  deploy          = (x)     install x; net.deploy[x]
}
MSink(p)      = {
  forward         = (x,y)   log_position_and_value[x,y]
```

```
}
[net.deploy[{
  sample        = ()    net.sample[];
                        install {sample = () sense (x) in net.forward[p, x];
                                               this.sample[]};
                        this.sample[]
  forward       = (x, y) net.forward[x, y]
 }];
 net.sample[], MSink(p)]^{p,r}_{b} |
[idle, MSensor(p_1)]^{p_1,r_1}_{b_1} | ... | [idle, MSensor(p_n)]^{p_n,r_n}_{b_n}
```

Notice that the implementation of the method sample has changed. Here, when the method is executed for the first time at each sensor, it starts by propagating the call to its neighborhood and then, it changes itself through an **install** call. The newly installed code of sample is the same as the one in the first implementation of the example. The method then continues to execute and calls the new version of sample, which starts sampling the field and forwarding values.

E. Sealing sensors

This example shows how we can install a sensor network with a module that contains a method, seal, that prevents any further dynamic re-programming of the sensors, preventing anyone from tampering with the installed code. The module also contains a method, unseal that restores the original deploy method, thus allowing dynamic re-programming again. The sink just installs the module containning these methods in the network by broadcasting a method call to deploy. Each sensor that receives the call, installs the module and floods the neighborhood with a replica of the call. Another message by the sink then replaces the deploy method itself and re-implements it to **idle**. This prevents any further instalation of software in the sensors and thus effectively seals the network from external interaction other than the one allowed by the remainder of the methods in the modules of the sensors.

```
MSensor       = {
  deploy      = (x) install x; net.deploy[x]
}
MSink         = { }
[net.deploy[{
  seal        = ()  install {deploy = ()  idle}
  unseal      = ()  install {deploy = (x) install x; net.deploy[x]}
```

```
  }];
  net.seal[], MSink]_b^{p,r} |
[idle, MSensor]_{b_1}^{p_1,r_1} | ... | [idle, MSensor]_{b_n}^{p_n,r_n}
```

IV. Discussion

In the previous sections, we focused our attention on the programming issues of a sensor network and presented a core calculus that is expressive enough to model fundamental operations such as local broadcast of messages, local sensing of the environment, and software module updates. CSN allows the global modeling of sensor networks in the sense that it allows us to design and implement sensor network applications as large-scale distributed applications, rather than giving the programmer a sensor-by-sensor view of the programming task. It also provides the tools to manage running sensor networks, namely through the use of the software deployment capabilities.

There are other important features of sensor networks that we consciously left out of CSN. In the sequel we discuss some of these features and sketch some ideas of how we would include support for them.

a) State: From a programming point of view, adding state to sensors is essential. Sensors have some limited computational capabilities and may perform some data processing before sending it to the sink. This processing assumes that the sensor is capable of buffering data and thus maintain some state. In a way, CSN sensors have state. Indeed, the atributes p, b, and r may be viewed as sensor state. Since these are characteristic of each sensor and are usually controlled at the hardware level, we chose to represent this state as parameters of the sensors. The programmer may read these values at any time through builtin method calls but any change to this data is performed transparently for the programmer by the hardware or operating system. As we mentioned before, it is clear that the value of b changes with time. The position p may also change with time if we envision our sensors endowed with some form of mobility (*e.g.*, sensors dropped in the atmosphere or flowing in the ocean).

To allow for a more systematic extension of our sensors with state variables we can assume that each sensor has a heap H where the values of these variables are stored: $[H, P, M]_b^{p,r}$. The model chosen for this heap is orthogonal to our sensor calculus and for this discussion we assume that we enrich the values v of the language with a set of *keys*, ranged over by k. Our heap may thus be defined as a map H from $keys$ into $values$. Intuitively, we can think of it as an associative memory with the usual built-in operations **put**, **get**, **lookup**, and **hash**. Programs running in the sensors may share state by exchanging keys. We assume also that these operations are atomic and thus no race conditions can arise.

With this basic model for a heap we can re-implement the *Ping* example from Section III-A with *scoped flooding* thus eliminating echos by software. We do this by associating a unique key to each remote procedure call broadcast to the network. This key is created through the built-in **hash** function that takes as arguments the position p and the battery b of the sensor. Each sensor, after receiving a call to ping, propagates the call to its neighborhood and generates a new key to send, with its position and battery charge, in a forward call. Then, it stores the key in its heap to avoid forwarding its own forward call. On the other hand, each time a sensor receives a call to forward, it checks whether it has the key associated to the call in its heap. If so, it does nothing. If not, it forwards the call and stores the key in the heap, to avoid future re-transmission.

```
MSensor(p, b) = {
  ping       = ()          net.ping[];
                           let k = hash[p,b] in net.forward[p, b, k];
                           put[k,_]
  forward    = (x, y, k)   if (!lookup[k]) then
                              net.forward[x, y, k];
                              put[k,_]
}
MSink(p, b)    = {...}
[net.ping[], MSink]^{p,r}_{b} |
[idle, MSensor(p1,b1)]^{p1,r1}_{b1} | ... | [idle, MSensor(pn,bn)]^{pn,rn}_{bn}
```

b) Events: Another characteristic of sensors is their *modus operandi*. Some sensors sample the field as a result of instructions implemented in the software that controls them. Such is the case with CSN sensors. The programmer is responsible for controlling the sensing activity of the sensor network. It is of course possible for sensor nodes to be activated in different ways. For example, some may have their sensing routines implemented at hardware or operating system level and thus not directly controllable by the programmer. Such classes of sensor nodes tipically sample the field periodically and are activated when a given condition arises (*e.g.*, a temperature above or below a given threshold, the detection of CO_2 above a given threshold, the detection of a strong source of infrared light). The way in which certain environmental conditions or *events* can activate the sensor is by triggering the execution of a handler procedure that processes the event. Support for this kind of event-driven sensors in CSN could be achieved by assuming that each sensor has a builtin handler procedure, say handle, for such events. The handler procedure, when activated, receives the value of the field that triggered the event. Note that,

from the point of view of the sensor, the occurrence of such an event is equivalent to the deployment of a method invocation **this**.handle[v] in its processing core, where v is the field value associated with the event. The sensor has no control over this deployment, but may be programmed to react in different ways to these calls, by providing adequate implementations of the handle routine. The events could be included in the semantics given in Section II-C with the following rule:

$$[P, M]_b^{p,r} \rightarrow_F [\textbf{this}.\mathsf{handle}\,[F(p)] \mid P, M]_b^{p,r} \qquad \text{(R-EVENT)}$$

As in the case of the builtin method for code deployment, the handler could be programmed to change the behavior of the network in the presence of events. One could envision the default handler as handle = (x) **idle**, which ignores all events. Then, we could change this default behavior so that an event triggers an alarm that gets sent to the sink. A possible implementation of such a dynamic re-programming of the network default handlers can be seen in the code below.

```
MSensor(p)  = { handle  = (x) idle }
MSink(p)    = { handle  = (x) idle }
[net.deploy[{
     handle = (x)   net.alarm[p,x]
     alarm  = (x,y) net.alarm[x,y]
 }];
 install {alarm  = (x,y) sing_bell[x,y]},
 MSink(p)] b^{p,r} |
[idle, MSensor(p1)] b1^{p1,r1} | ... | [idle, MSensor(pn)] bn^{pn,rn}
```

where the default implementation of the handle procedure is superseded by one that eventualy triggers an alarm in the sink.

More complex behavior could be modeled for sensors that take multiple readings, with a handler associated with each event.

c) Security: Finally, another issue that is of outmost importance in the management of sensor networks is security. It is important to note that many potential applications of sensor networks are in high risk situations. Examples may be the monitorization of ecological disaster areas, volcanic or sismic activity, and radiation levels in contaminated areas. Secure access to data is fundamental to establish its credibility and for correctly assessing risks in the management of such episodes. In CSN we have not taken security issues into consideration. This was not our goal at this time. However, one feature of the calculus may provide interesting solutions for the future. In fact, in CSN, all computation within a sensor

results from an invocation of methods in the modules of a sensor, either originating in the network or from within the sensor. In a sense the modules M of the sensor work as a firewall that can be used to control incomming messages and implement security protocols. Thus, all remote method invocations and software updates might first be validated locally with methods of the sensor's modules and only then the actions would be performed. The idea of equipping sensors, or in general *domains*, with some kind of membrane that filters all the interactions with the surrounding network has been explored, for instance, in [**?**], in the M-calculus [**?**], in the Kell calculus [**?**], in the Brane calculi [**?**], in Miko [**?**], and in [**?**]. One possible development is to incorporate some features of the *membrane model* into CSN. The current formulation of the calculus also assumes that all methods in the module M of a sensor $[P, M]_b^{p,r}$ are visible from the network. It is possible to implement an access policy to methods in such a way that some methods are private to the sensor, *i.e.*, can only be invoked from within the sensor. This allows, for example, the complete encapsulation of the state of the sensor.

V. Conclusions and Future Work

Aiming at providing large-scale sensor networks with a rigorous and adequate programming model (upon which operating systems and high-level programming languages can be built), we presented CSN — a Calculus for Sensor Networks, developed specifically for this class of distributed systems.

After identifying the necessary sensing, processing, and wireless broadcasting features of the calculus, we opted to base our work on a top-layer abstraction of physical and link layer communication issues (in contrast with previous work on wireless network calculi [**?**, **?**]), thus focusing on the system requirements for programming network-wide applications. This approach resulted in the CSN syntax and semantics, whose expressiveness we illustrated through a series of implementations of typical operations in sensor networks. Also included was a detailed discussion of possible extensions to CSN to account for other important properties of sensors such as state, sampling strategies, and security.

As part of our ongoing efforts, we are currently using CSN to establish a mathematical framework for reasoning about sensor networks. One major objective of this work consists in providing formal proofs of correctness for data gathering protocols that are commonly used in current sensor networks and whose performance and reliability has so far only been evaluated through computer simulations and ad-hoc experiments.

From a more practical point of view, the focus will be set on the development of a prototype implementation of CSN. This prototype will be used to emulate the behavior of sensor networks by software and, ultimately, to port the programming model to a natural development architecture for sensor

network applications.

ACKNOWLEDGEMENTS

The authors gratefully acknowledge insightful discussions with Gerhard Maierbacher (Departamento de Ciência de Computadores, Faculdade de Ciências, Universidade do Porto).

This figure "reachback.jpg" is available in "jpg" format from:

http://arXiv.org/ps/cs/0612093v1

This figure "sensornet.jpg" is available in "jpg" format from:

http://arXiv.org/ps/cs/0612093v1